Machine Characterization Based on an Abstract High-Level Language Machine

RAFAEL H. SAAVEDRA-BARRERA, STUDENT MEMBER, IEEE, ALAN JAY SMITH, FELLOW, IEEE, AND EUGENE MIYA, MEMBER, IEEE

Abstract—**Runs of a benchmark or a suite of benchmarks are inadequate to either characterize a given machine or to predict the running time of some benchmark not included in the suite. Furthermore, the observed results are quite sensitive to the nature of the benchmarks, and the relative performance of two machines can vary greatly depending on the benchmarks used. In this paper, we report on a new approach to benchmarking and machine characterization. The idea is to create and use a machine characterizer, which measures the performance of a given system in terms of a Fortran abstract machine. Fortran is used because of its relative simplicity and its wide use for scientific computation. The analyzer yields a set of parameters which characterize the system and spotlight its strong and weak points; each parameter provides the execution time for some primitive operation in Fortran.**

We present measurements for a large number of machines ranging from small workstations to supercomputers. We then combine these measurements into groups of parameters which relate to specific aspects of the machine implementation, and use these groups to provide overall machine characterizations. We also define the concept of pershapes, which represent the level of performance of a machine for different types of computation. We introduce a metric based on pershapes that provides a quantitative way of measuring how similar two machines are in terms of their performance distributions. This metric is related to the extent to which pairs of machines have varying relative performance levels depending on which benchmark is used.

Index Terms—**Abstract high-level language machine, benchmarking, execution time prediction, machine characterization, performance distance between machines, performance shapes, program statistics.**

I. INTRODUCTION

ONE approach to comparing the CPU performance of different machines is to run a set of benchmarks on each. Benchmarking has the advantage that since real programs are being run on real machines, the results are valid, at least for that set of benchmarks; such results are much more believable than estimates produced from models of the system, no matter how detailed. To the extent that the benchmark set is representative of some target workload, the observed performance differences will reflect differences in practice.

Considerable effort has been expended to develop benchmark suites that are considered to reflect real workloads. Among them are the Livermore Loops [28], the NAS kernels [1], [2], and synthetic benchmarks (e.g., Dhrystone [40], [41], Whetstone [12]). Unfortunately, there are a number of shortcomings to benchmarking [15], [42]. 1) It is very difficult to explain the benchmark results from the characteristics of the machines. 2) It is not clear how to combine individual measurements to obtain a meaningful evaluation of the various systems. 3) Given that there is almost never a good model of the machines being benchmarked, it is not possible to validate the results, nor to make predictions and/or extrapolations about expected performance for other programs. 4) Unless the benchmarks are tuned for each machine architecture, they may not take advantage of important architectural features. 5) The large variability in the performance of highly optimized computers is difficult to characterize with benchmarks. For example, using benchmarks, Harms *et al.* found that the relative performance between the Fujitsu VP-200 and the CRAY X-MP/22 varied from 0.41 to 5.39 on individual programs [20]; the ratio for the whole workload was only 1.12.

In this research, we present a new approach to characterizing machine performance. We do this via "narrow spectrum" benchmarking, by which we measure the performance of a machine on a large number of very specific operations, in our case, primitive operations in Fortran. This set of measurements characterizes each specific CPU. We separately analyze specific programs, ignoring at this stage of our research compiler optimizations and vector instructions. We can then combine the frequency of the primitive operations with their running times on various machines to predict the running time of any analyzed program on any analyzed machine. This approach also gives us considerable insight into both the machines and the programs, since the effects of individual parameters are immediately evident.

This paper provides an overview of this approach to performance evaluation, but concentrates on the specific issue of machine characterization; prediction of the execution time of benchmarks is done in [34] and [36]. Section II gives a somewhat more detailed overview of our research. In Section

Manuscript received March 1, 1989; revised July 15, 1989. This work is based on research supported in part by NASA under Consortium Agreement NCA-128 and Cooperative Agreement NCC2-550, the Mexican Council for Science and Technology (CONACYT) under Contract 49492, the National Science Foundation under Grant MIP-8713274, by the State of California under the MICRO program, and by the International Business Machines Corporation, Digital Equipment Corporation, Tandem, Hewlett-Packard, Philips Research Laboratories/Signetics, and Apple Computer Corporation.

R. H. Saavedra-Barrera and A. J. Smith are with the Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

E. Miya is with NASA Ames Research Center, Moffett Field, Mountain View, CA.

IEEE Log Number 8931177.

III, we describe the program analyzer and also the execution predictor. The parameters used to characterize a machine are explained in Section IV. The methodology used to make measurements is presented in Section V, and the parameters derived from a number of machines are given in Section VI. A comparison of machines is also provided in that section. The concepts of performance distributions (pershapes) and pershape distances between machines are given in Section VII. Some unresolved issues are considered in Section VIII.

II. System Characterization and Performance Evaluation

The idea behind our approach is to distinguish between two different activities often ignored in machine evaluation; these are system characterization and performance evaluation. We define *system characterization* as an n-value vector where each component represents the performance of a particular operation (P_i). This vector ($\langle P_1, P_2, \cdots, P_m \rangle$) fully describes the system at some level of abstraction. The parameters we use are a set of primitive operations, as found in the Fortran programming language, and are defined in Section IV. We measure the values of the parameters using a *system characterizer*, which runs a set of "software experiments," which detect, isolate, and measure the performance of each basic operation. This approach is similar to studies which use a low-level machine architecture based model [32], but we use a higher level machine model.

The *performance evaluation* of a group of systems is the measurement of some number of properties during the execution of some workload. One property may be the total execution time to complete some job. It is important to note that the results depend, and are only valid, for the set of programs used in the evaluation, and are sensitive to not only the machine, but also the compiler, the operating system, and the libraries. In this research, we focus on the execution time of computationally intensive programs as our metric for evaluating different architectures.

A. A Linear Model for Program Execution

Our research is based on the assumption that the execution time of a program can be partitioned into independent time intervals, each corresponding to the execution of some operation of an abstract Fortran machine (AFM). The AFM is the same for all machines, independent of the hardware; only the times for the Fortran operations differ. As is shown in [34], and to a lesser extent later in this paper, this assumption is reasonably accurate.

A similar approach is often used at the machine instruction level to evaluate implementations of an instruction set architecture and design the next one [27], [32]. In that case, instruction execution time is modeled as a linear sum of operations that take time (instruction execution times, pipeline interlocks, and storage access delays) weighted by their frequencies. Our time consuming operations are Fortran source statements; the implementation of such operations will differ between machines, and even when compilers or optimization levels are changed.

Our model of the total execution time is the following. Let $\boldsymbol{P_M} = \langle P_1, P_2, \cdots, P_n \rangle$ be the set of parameters that characterize the performance of machine M. Let $\boldsymbol{C_A} = \langle C_1, C_2, \cdots, C_n \rangle$ be the normalized dynamic distribution of operations in program A, and let C_{total} denote the total number of operations executed in program A. We obtain the expected execution time of program A on machine M

$$T_{A,M} = C_{\text{total}} \sum_{i=1}^{n} C_i P_i = C_{\text{total}} \boldsymbol{C_A} \cdot \boldsymbol{P_M} \qquad (1)$$

where

$$\sum_{i=1}^{n} C_i = 1.$$

In general, given machines $M_1, M_2, \cdots, M_m$, with characterizations $\boldsymbol{P_{M_1}}, \boldsymbol{P_{M_2}}, \cdots, \boldsymbol{P_{M_m}}$, and a workload $\boldsymbol{W}$ formed by programs $A_1, A_2, \cdots, A_l$ with dynamic distributions $\boldsymbol{C_{A_1}}, \boldsymbol{C_{A_2}}, \cdots, \boldsymbol{C_{A_l}}$, the expected execution time of machine M_k on workload $\boldsymbol{W}$ is

$$T_{W,M_k} = \sum_{j=1}^{l} C_{\text{total}_A} \boldsymbol{C_{A_j}} \cdot \boldsymbol{P_{M_k}} \qquad (2)$$

where C_{total_A} is the total number of operations executed in program A_j. T_{W,M_i} provides a way to make a direct comparison between several machines with respect to workload $\boldsymbol{W}$.

Using this model it is possible not only to compare two different machine architectures using any workload, but also to explain their results in terms of the abstract parameters. Let $\Phi_{M,A} = \langle \phi_1, \phi_2, \cdots, \phi_n \rangle$ be the normalized distribution of the execution time for program A executed on machine M. Define

$$\phi_i = \frac{C_{A,i} P_{M_{k,i}}}{\boldsymbol{C_A} \cdot \boldsymbol{P_M}}.$$

Vector $\Phi_{M,A}$ decomposes the total execution time in terms of each parameter and makes it possible to identify which operations are the most time consuming. We would expect that different machines will have different distributions, even for different implementation of the same architecture or/and different compilers. Once we have the machine characterizations, it is possible to study the effect of changes in the normalized dynamic distribution without writing real programs that correspond to these distributions, and in this way detect which parameters have a significant impact in the execution time for some machines.

An advantage of this scheme is that the $l \cdot m$ machine-program combinations only require that each machine be measured once to obtain its characterization, and also that each program be analyzed once ($l + m$). Moreover, once the machine has been measured, its characterization can be used at any time in the future for additional evaluations, in contrast to benchmarking in which access to the machine (same model, operating system, compiler, libraries) is needed for each new set of benchmarks.

B. Limits of the Linear Model

The only way in which the linear model can give acceptable results is if the following conditions hold. 1) The experimental measurements are representative of "typical" occurrences of the parameters in real programs. 2) The errors caused by the low resolution and the intrusiveness of the measuring tools are small compared to the magnitude of the measurements. 3) Variability in the execution mean time caused by data dependencies, external concurrent activity, and nonreproducible conditions is small, and therefore does not significantly affect the results. In some cases, the above conditions cannot be satisfied, especially in highly pipelined machines where the execution time when there is a register dependency conflict is several times greater than the execution time without this delay. An example of this is the CYBER 205, where an add or multiply can take as little as 20 ns to execute, when the pipeline is full, or as much as 100 ns in the worst case [21]. If we consider the following two statements

$$X9 = ((X1 + X2) * (X3 + X4)) + ((X5 + X6) * (X7 + X8))$$

$$X6 = ((X1 + X2) * X3 + X4) * X5$$

we find that the execution of the first statement takes approximately 360 ns, while the execution time of the second takes 400 ns. A simple linear model will estimate that the execution of the second statement will be less than that of the first statement, unless the model contains information on how the execution time is affected by data dependencies. Branching and interrupts also prevent the pipeline from working at peak speed. Although it is difficult to detect and measure how each machine will execute different statements, it is always possible to create new parameters that take into account data dependencies and measure the extra penalty in the execution time. In practice, the number of parameters cannot be expanded indefinitely.

C. Fortran and Other Programming Languages

The model presented above can also be applied to other general purpose languages. We chose Fortran instead of other programming languages for the following reasons: 1) most large-scale scientific computation, accounting for most of the CPU time on supercomputers, is done in Fortran; 2) the number of language constructs in Fortran is small; and 3) the execution time of most of the operations in Fortran does not depend on the value of the arguments. It is therefore natural to experiment first with a less complex programming language and test whether it is possible to make acceptable predictions. Most of the differences between Fortran and other general purpose languages do not prevent building an abstract machine model, although a model with a larger number of parameters and better experiments would be required.

D. Compiler Optimization

Thus far, we have presented our linear model as if a given source program were simply and directly translated into the corresponding machine code. In reality, compilers optimize the code, even when the optimizer is nominally inactive (optimization level 0). Including the effect of optimizations in our analysis is a difficult problem and is currently under study; we summarize some of the issues in this section.

The problem of evaluating the effects of compiler optimizers can be broken down into three subproblems: 1) the experimental detection of which optimizations can be applied by the optimizer, i.e., what can the compiler optimizer do; 2) the measurement of the performance improvement that a particular optimization will produce in a program; 3) the measurement of the possible optimizations present in the source code. Each is briefly discussed below.

The first point is similar to machine characterization, but instead of measuring the performance of some operation, we are interested in detecting which optimizations can be applied by the compiler and in which cases. This may appear as an easy task, but unfortunately, in many compilers, optimizations are implemented ad hoc, e.g., a given transformation may be used for only one data type and not another, although the transformation is type independent [25], [26]. Reference [5] presents an attempt to detect optimizations in the domain of a vectorizing compiler.

The second subproblem deals with the quantification of the performance benefits of individual transformations. Several studies have tried to measure the performance effect of some algorithms for code improvement [11], [39], [10], [3], [22], [33]. Most have been carried out in the process of engineering an optimizing compiler and performance evaluation has not been its main driving force; [33] is an exception. In addition, some of the studies have used a set of small programs with a very regular structure like matrix multiplication, FFT, Baskett puzzle, etc., in which the execution time is significantly reduced by applying one or two transformations. It is necessary that we develop benchmarks to measure the effect of individual optimizations in large and realistic applications using several optimizing compilers.

The third problem refers to the amount of optimizable code present in real applications. Detecting which optimizations are done by optimizers and measuring their performance effects is only part of the problem; we also need to know the extent to which programs contain optimizable code. The measurement of optimization opportunities in programs could potentially be done by modifying an existing highly optimizing compiler.

Several problems arise when we deal with compiler optimizers and attempt to measure their performance effect. The first is that most optimizations cannot be measured in isolation. It is common that the opportunity to apply some optimization is the result of a previous transformation, and in some cases the first transformation may not necessarily improve the execution time of the programs. One example is in-line substitution of leaf procedures. The benefits of this optimization (by eliminating the procedure call) are normally less than the benefits of other optimizations that are exposed by in-line substitution. This is especially true in benchmarks like Linpack and Dhrystone.

A second problem is machine-dependent optimizations. Proposing a general framework for the evaluation of opti-

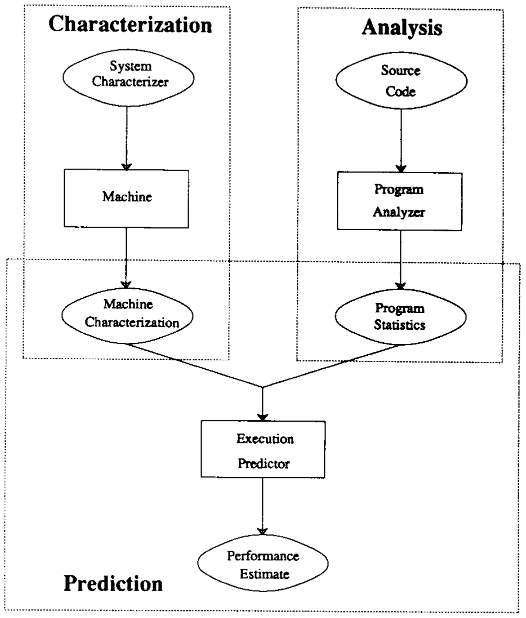


Fig. 1. The process of characterization, analysis, and prediction.

mizers hampers our ability to evaluate machine-dependent optimizations. Most of these optimizations do not have an equivalent in other architectures, and it may be argued that these transformations are not really optimizations of the source code, but deal with the efficient use of the machine resources. However, some attempt should be made to measure the effect of machine-dependent optimizations and compare their effectiveness against those that are machine independent.

Most of our current efforts are directed at point one above, in an attempt to evaluate the effectiveness of various compilers in optimizing code.

III. Description of the System

In the last section, we showed what we need in order to characterize machines using the linear model and how to use this information to make predictions about the execution time of programs. We have implemented 1) a system characterizer and assembled a library of machine characterizations (P_M); 2) a program analyzer that generates the dynamic distribution (C_A) and the total number of operations (C_{total}) of Fortran programs; and 3) an execution predictor that takes P_M, C_A, and C_{total} and estimates the expected execution time of the applications. The complete process of characterization, analysis, and prediction is shown in Fig. 1. In the next two subsections, we give an overview of the program analyzer and the execution predictor. A more in-depth presentation of the system characterizer follows.

A. Program Analyzer

The program analyzer (PA) decomposes Fortran programs statically and dynamically in terms of the abstract parameters. In addition, both models, the performance model associated with machines and the execution model associated with the applications, are identical. Thus, it is possible using the dynamic distribution to compare different programs, putting the emphasis not on their syntactic or semantic properties, but in how they affect the performance of different systems. The dynamic statistics are independent of the code generated by each compiler, and they only depend on the source code and the data used in the execution.

The PA is basically the front end of a Fortran compiler. It takes as its input a Fortran program and after making a lexical and syntactical analysis, it outputs an instrumented version of the original program, from which we obtain the dynamic statistics. In addition, the PA also gives the static statistics of each parameter for each basic block.

B. Execution Predictor

The execution predictor (EP) combines the machine characterization P_M with the dynamic statistics C_A to obtain estimates of the expected execution time of programs, as indicated in (1). The execution time is computed for each statement of the program, and this makes it possible to compute estimates for different parts of the program. In addition to the expected execution time, the EP also reports the variance of the estimate and the expected execution time per parameter along with its variance.

1) Accuracy of the Execution Predictor: The correct test for any model is its ability to predict the behavior of the system it is trying to model. Unfortunately, many performance evaluation techniques, like benchmarking, completely ignore prediction and thus can only be considered as observations of the phenomena.

The use of our model for prediction is discussed in [34] and in a forthcoming paper [36]; here we just present a small sample of our results. To test the predictive ability of our model, we selected a suite of 9 programs ranging from small benchmarks that execute for a couple of seconds on a Sun 3/50 to large applications which required several hours of CPU time. The set of machines selected went from small workstations like the RT-PC to supercomputers like the CYBER 205 and the CRAY X-MP/48. We predicted execution times for all the program and system combinations and compared these to 91 measurements of real execution times. The results were very encouraging: 62 percent of the predictions were correct within 10 percent, 86 percent were within 15 percent, and 97 percent were within 20 percent of the correct running time. The root mean squared error across machines varied between 8.2 and 13.9 percent, and the same metric across programs varied from 4.5 to 15.9 percent. Table I shows the measured and predicted running times for the VAX-11/780, and as may be evident, they are quite close. The accuracy of our predictions supports our confidence that an abstract model of a system at the programming language level provides a good general framework for uniform system comparison.

IV. The Fortran Abstract Machine

By using a common parametric model, we are able to compare the performance of different architectures and make a fair comparison between them with respect to their execution of Fortran programs. An extremely important issue is that of selecting which and how many parameters. Increasing the

TABLE I
EXECUTION ESTIMATES AND ACTUAL RUNNING TIMES FOR THE VAX-11/780. REAL TIME AND PREDICTION IN SECONDS; ERRORS IN PERCENTAGE

program	prediction (sec)	real time (sec)	error (%)
Alamos	1702.7	1581.7	+7.65
Baskett	16.17	14.85	+8.99
Erathostenes	2.642	2.766	-10.99
Linpack	227.5	220.1	+3.38
Livermore	653.5	611.0	+6.96
Mandelbrot	32.13	33.42	-3.86
Shell	8.803	9.183	-4.14
Smith	1018.8	1087.5	-6.32
Whetstone	21.74	21.57	+0.79
average			+0.26
root mean sq.			9.52

number of parameters yields increased accuracy, up until the point where measurement errors and nonlinearities in the real system (i.e., the linear model is not perfectly accurate) dominate. We are also limited in our accuracy by the fact that we do not analyze the code generated by the compiler, nor use information that is not contained in the source code of the program. In the next subsection, we present the set of parameters used in our model and give a brief description of what they measure. A more extensive discussion can be found in [34].

A. Parameters in the System Characterizer

Each parameter of the model can be classified in one of the following broad categories: arithmetic and logical, procedure calls, array references, branching and iteration, and intrinsic functions. We decided which parameters to include in our model in an iterative manner. Initially we associated parameters with obvious basic operations, and after a first version of the system was running, new parameters were incorporated to distinguish between different uses and execution times of the "same" abstract operation in the program. This was mainly the result of detecting a significant error between our predictions and real execution times. Thus, the number of parameters has increased from 76 in [34] to 102 currently. Although every basic operation in Fortran is characterized by some parameter, we have made simplifications for operations which were rarely executed in the benchmarks we used. It is straightforward to include new parameters in the model, and to write new experiments for the system characterizer. The parameters are classified in 18 different groups according to the semantics of the operation; Tables II and III present the parameters.

Fortran is a language for scientific and numeric applications. For this reason most of the parameters deal with arithmetic, logical, or trigonometric functions. In addition to the arithmetic operators, Fortran also provides six relational and six logical operators. Tables II and III (groups 1-8) show the 56 arithmetic parameters grouped by data type (real, complex, or integer), size (single or double precision) and storage class (local or global), and the logical and conditional parameters (groups 9–10). (Each arithmetic mnemonic is formed by appending the first letter of the operation with the first letter of the data type (*R*, *C*, or *I*), plus a letter identifying the size of the operands (*S* or *D*) and finally the first letter of the storage class (*L* or *G*). Four major arithmetic operators are represented: addition (includes subtraction), multiplication, division (quotient), and exponentiation. Addition between an array index and a constant is treated as a special parameter (74 IADD); most compilers compute the result and eliminate the addition. In the case of exponentiation and with a real base we distinguish two cases: one when the exponent is real and

TABLE II
THE SET OF PARAMETERS MEASURED BY THE SYSTEM CHARACTERIZER (PART 1 OF 2). ARITHMETIC OPERATIONS ARE CLASSIFIED TAKING INTO ACCOUNT THE TYPE, WIDTH, AND STORAGE CLASS OF THEIR OPERANDS

Parameters in the system characterizer (part 1 of 2)

1 real operations (single, local)		5 real operations (single, global)	
01 SRSL	store	29 SRSG	store
02 ARSL	addition	30 ARSG	addition
03 MRSL	multiplication	31 MRSG	multiplication
04 DRSL	division	32 DRSG	division
05 ERSL	exponential (X^I)	33 ERSG	exponential (X^I)
06 XRSL	exponential (X^Y)	34 XRSG	exponential (X^Y)
07 TRSL	memory transfer	35 TRSG	memory transfer

2 complex operations, local operands		6 complex operations, global operands	
08 SCSL	store	36 SCSG	store
09 ACSL	addition	37 ACSG	addition
10 MCSL	multiplication	38 MCSG	multiplication
11 DCSL	division	39 DCSG	division
12 ECSL	exponential (X^I)	40 ECSG	exponential (X^I)
13 XCSL	exponential (X^Y)	41 XCSG	exponential (X^Y)
14 TCSL	memory transfer	42 TCSG	memory transfer

3 integer operations, local operands		7 integer operations, global operands	
15 SISL	store	43 SISG	store
16 AISL	addition	44 AISG	addition
17 MISL	multiplication	45 MISG	multiplication
18 DISL	division	46 DISG	division
19 EISL	exponential (I^2)	47 EISG	exponential (I^2)
20 XISL	exponential (I^J)	48 XISG	exponential (I^J)
21 TISL	memory transfer	49 TISG	memory transfer

4 real operations (double, local)		8 real operations (double, global)	
22 SRDL	store	50 SRDG	store
23 ARDL	addition	51 ARDG	addition
24 MRDL	multiplication	52 MRDG	multiplication
25 DRDL	division	53 DRDG	division
26 ERDL	exponential (X^I)	54 ERDG	exponential (X^I)
27 XRDL	exponential (X^Y)	55 XRDG	exponential (X^Y)
28 TRDL	memory transfer	56 TRDG	memory transfer

TABLE III
THE SET OF PARAMETERS MEASURED BY THE SYSTEM CHARACTERIZER (PART 2 OF 2). EACH STANDARD INTRINSIC FUNCTION IN FORTRAN IS REPRESENTED BY ONE PARAMETER. FOR EXAMPLE, SINE AND COSINE ARE CHARACTERIZED BY THE SAME PARAMETER SINx, WHERE x CAN BE S, D, OR C

Parameters in the system characterizer (part 2 of 2)

9 logical operations (local)		10 logical operations (global)	
57 ANDL	AND & OR	62 ANDG	AND & OR
58 CRSL	compare, real, single	63 CRSG	compare, real, single
59 CCSL	compare, complex	64 CCSG	compare, real, double
60 CISL	compare, integer, single	65 CISG	compare, integer, single
61 CRDL	compare, real, double	66 CRDG	compare, real, double

11 function call and arguments		13 branching parameters	
67 PROC	procedure call	69 GOTO	simple goto
68 AGRS	argument load	70 GCOM	computed goto

12 references to array elements		14 DO loop parameters	
71 ARR1	array 1 dimension	75 LOIN	loop initialization (step 1)
72 ARR2	array 2 dimensions	76 LOOV	loop overhead (step 1)
73 ARR3	array 3 dimensions	77 LOIX	loop initialization (step n)
74 IADD	array index addition	78 LOOX	loop overhead (step n)

15 intrinsic functions (real)		16 intrinsic functions (double)	
79 LOGS	logarithm	87 LOGD	logarithm
80 EXPS	exponential	88 EXPD	exponential
81 SINS	sine	89 SIND	sine
82 TANS	tangent	90 TAND	tangent
83 SQRS	square root	91 SQRD	square root
84 ABSS	absolute value	92 ABSD	absolute value
85 MODS	module	93 MODD	module
86 MAXS	max. and min.	94 MAXD	max. and min.

17 intrinsic functions (integer)		18 intrinsic functions (complex)	
95 ABSI	absolute value	98 LOGC	logarithm
96 MODI	module	99 EXPC	exponential
97 MAXI	max. and min.	100 SINC	sine
		101 SQRC	square root
		102 ABSC	absolute value

the other when it is an integer. Different algorithms are used to implement them. In the case of an integer base, there are also two cases determined by the magnitude of the exponent: one case for an exponent equal to two and the other when it is greater than two. Given the small number of exponentiations executed, these simplifications are sufficient.

All the arithmetic operations in the store (store to memory) category (arithmetic parameters with first letter *S*) correspond to the time it takes to store the result of an expression. In Fortran, loading an operand is not an independent operation and therefore the execution time for most loads is included in the arithmetic parameter. The exception is for an assignment statement which does not involve the execution of any operation (memory to memory). The memory transfer parameter (first letter *T*) represents the execution time of transfer between variables.

B. Additional Parameters

Procedures and functions are characterized by two parameters (group 11). One parameter measures the joint execution of the prologue and epilogue of the call (PROC). A second parameter measures the time it takes to load the address of each argument (in Fortran arguments are passed by reference) either into the registers, the static environment of the callee subprogram, or into the execution stack; different architectures use different protocols.

Although Fortran has three different types of GO TO statements: unconditional, assigned, and computed, we only need two parameters to cover all the cases (group 13). The only distinction is between a direct jump (GOTO) and a computed jump (GCOM). In the first case, the address is known by the compiler and a direct jump is used, while on the second the target depends on the value of an expression and it is computed at run time.

For DO loops we can identify two sources of overhead: the time to initialize the loop and the time to update the index and compare it against the limit. In addition, some machines implement loops with unit step differently from nonunit step loops. In some cases, the loop is transformed to a loop with a unit step, which sometimes increases the overhead of nonunit loops. Four parameters (LOIN, LOOV, LOIX, LOOX) characterize loops with unit and nonunit increment (group 14).

Although Fortran has three different types of IF statements, neither of these have been found to need special parameters for their characterization. The block IF and the logical IF are decomposed in two parts: the evaluation of the predicate (arithmetic–logical expression) and a direct branch. The arithmetic IF is different only in the way it branches. We handle the branching part as a computed GOTO (GCOM).

Array variables in expressions are treated as ordinary variables plus an additional overhead to compute the address of the element. We have three parameters (ARR1, ARR2, and ARR3) that characterize the dimension of an array reference (group 12). The overhead for variables in four and five dimensions is computed in the execution predictor using a linear combination of the three basic parameters. We found that most of the applications we examined had very few arrays with more than three dimensions and no examples of more than five. As we mentioned in Section IV-A, adding a constant to an index variable is considered different from an ordinary integer add. The parameter IADD is used for this special case.

Intrinsic functions are represented by 24 parameters corresponding to the functions most used in scientific programs (groups 15–18). Although the execution time of an intrinsic function is normally a function of the arguments and not constant, we have assumed that it is constant. This is because the execution frequency of these functions is generally low, the arguments unpredictable, and we have found that our assumption of constant execution time is a good enough approximation.

C. Global Versus Local Variables

Most operators are characterized by several parameters, depending on the operand types and sizes, and storage class (common/local). Global variables in Fortran (COMMON) are sometimes treated differently from local variables. In some compilers, variables stored in COMMONs are treated as components of a structure using a base-descriptor for each COMMON block which points to the first element of the COMMON. An operand is loaded by first adding an offset to the base-descriptor and then loading the operand. This way of treating simple variables increases their load time.

V. Machine Characterizer

The machine characterizer (MC) consists of 102 "software experiments" that measure the performance of each individual parameter needed to completely characterize a Fortran machine (see Tables II and III). The MC is written as a Fortran program and runs from 200 s, on a machine with good clock resolution, to 2000 s on machines with 1/60th second resolution. We have run the MC on many different machines ranging from low-end workstations to supercomputers. Each experiment tries to measure the execution time that each parameter takes to execute in "typical" Fortran programs. This "typical" execution time was obtained by looking at real programs and also by modifying those experiments that were identified as generating the biggest error in our predictions.

A. Experiment Structure

Timing a benchmark is very different from making a detailed measurement of the parameters in the system characterizer. For some benchmarks the system clock is enough for timing purposes, and repetition of the measurements normally yields an insignificant variance in the averaged results. On the other hand, the measurement of the parameters in the system characterizer is more difficult due to a number of factors:

- the short execution time of most operations (5 ns–10 μs)
- the resolution of the measuring tools (≥ 1 μs)
- the difficulty of isolating the parameters using a program written in Fortran
- the intrusiveness of the measuring tools
- variations in the hit ratio of the memory cache
- external events like interrupts, multiprogramming, and I/O activity
- the need to obtain repeatable results and accuracy.

```
      LIMIT = LIMIT0 * SPEEDUP * (TMAX - TMIN) / 2.
      DO 4 K = 1, REPEAT
1       COUNTER = 1
        TIME0 = SECOND ()
2       IF (COUNTER .GT. LIMIT)  GO TO 3
          ...
          body of the test
          ...
        COUNTER = COUNTER + 1
        GO TO 2
3       TIME1 = SECOND ()
        IF (TIME1 - TIME0 .GE. TMIN .AND. TIME1 - TIME0 .LE. TMAX)  GO TO 4
        LIMIT = .5 * LIMIT * (TMAX - TMIN) / (TIME1 - TIME0)
        GO TO 1
4     SAMPLE(K) =  (TIME1 - TIME0) / LIMIT
      CALL STAT (REPEAT, SAMPLE, AVE, VAR)
```

Fig. 2. The basic structure of an experiment. The statement "IF(TIME1 − TIME0, · · ·" enforces the execution of each test for more than TMIN and less than TMAX seconds. If the execution is outside this interval, a new value of LIMIT is computed and the test is repeated.

Most of our primitive operations have execution times of from ten to thousands of nanoseconds and are implemented with a single or a small number of machine instructions. For this reason, direct measurement is not possible, especially since our tests should work for many different architectures. Furthermore, the need to isolate an operation for measurement normally requires robust tests to avoid optimizations[1] from the compiler that would eliminate the operation from the test and distort the results [6]. Different techniques must be used, in particular avoiding the use of constants inside the test loops; using IF and GO TO instructions instead of the DO LOOP statements to control the execution of the test; and initializing variables in external procedures to avoid constant folding. Separate compilation of variable initialization procedures is used to make sure that the body of the test does not give enough information to the compiler to eliminate the operation being measured from inside the test loop.

B. Test Structure and Measurement

The measurement tools we have are the system clock and the repeated execution of a sequence of statements. The resolution of the clock, the overhead of the timing routine, and the overhead of the statements that control of measurements are the sources of error that we can control or work around. Variations in the hit ratio of the cache, interrupts, multiprogramming and I/O activity are more difficult to eliminate and measure (see Section VIII).

We use three different methods to measure the execution time of the parameters. The first is by *direct* measurement, i.e., executing some operation for some number of times and in different contexts. The second is with a *composite* measurement. In this case, we execute a number of different operations and subtract the execution time of the known parameters to obtain the value of the one that is unknown. The third possibility is with an *indirect* measurement. Some parameters of the model are "coupled;" it is not possible to execute one without executing the other. The way to measure one of the parameters is to run two or more tests with a different number of operations; the solution of a set of linear equations gives the correct result. Fig. 2 shows the basic structure of our tests. This same structure is used in all the tests.

[1] Even when we compile without optimization, compilers try to apply some standard optimization techniques, such as constant folding, short-circuiting of logical expressions, and computing the address of an element in an array.

The sequence of statements to measure corresponds to the "body of the test." These statements are executed for some number of times (LIMIT) and the execution time is measured (function SECOND). This time is called an observation. TMIN and TMAX control the minimum and maximum time that each observation should run (TMIN $\leq$ TIME1 − TIME0 $\leq$ TMAX). The two statements before the GO TO 1 enforce this condition. The DO loop is used to get several (REPEAT) observations to obtain a meaningful statistic. Because we do not know *a priori* how fast or slowly an operation executes in an arbitrary machine, we extrapolate by using the time it takes to run the test in the CRAY X-MP/48 and multiply by their relative speeds. This is done using LIMIT0, which is the number of times the test runs in the CRAY X-MP/48, and SPEEDUP that gives the relative speed of the machine. The relative speed is computed by running a small test at the beginning of the characterization.

C. Experimental Error and Confidence Intervals

There are many known sources for the variability of the CPU time [13], [29] and consequently in our measurements. Some of these factors are timer resolution of the clock, improper allocation of the CPU for I/O interrupt handling, cycle stealing, and changes in cache hit ratios due to interference with concurrent tasks. Small errors in the measurements have considerable impact in the predictions we make and we must measure and compensate for them.

We will proceed to derive expressions for the variance and the confidence intervals of the measurements. We know that each sample measurement is equal to

$$T_{j_1} - T_{j_0} = N_{\text{limit}_j}(B_j + IF_{\text{overhead}}) + C_{\text{overhead}} \tag{3}$$

where T_{j_1} and T_{j_0} correspond to TIME1 and TIME0 in Fig. 2; N_{limit_j} (LIMIT) is the number of times the body of the test (B_j) is executed, C_{overhead} is the overhead of the timing function, and IF_{overhead} represents the extra instructions that control the test. Using (3) we can compute the time to execute once the body of the test

$$B_j = \frac{T_{j_1} - T_{j_0} - C_{\text{overhead}}}{N_{\text{limit}_j}} - IF_{\text{overhead}}.$$

Now the mean time and variance for a sample of size N_{repeat} is

$$\hat{B} = \frac{1}{N_{\text{repeat}}} \sum_{j=1}^{N_{\text{repeat}}} B_j,$$

$$\sigma^2 B = \frac{1}{N_{\text{repeat}} - 1} \sum_{j=1}^{N_{\text{repeat}}} (B_j - \hat{B})^2. \tag{4}$$

To obtain the mean value of parameter $\hat{P}_i$ we need to know if the test is direct, composite, or indirect. Let N be the number of times parameter $\hat{P}_i$ is executed inside the body of the test, then the mean value and variance of parameter $\hat{P}_i$ in a direct test are

$$\hat{P}_i = \frac{\hat{B}}{N}, \qquad \sigma^2 P_i = \frac{\sigma^2 B}{N^2}. \tag{5}$$

In a composite test we have

$$\hat{P}_i = \frac{\hat{B} - \hat{W}_{\text{extra}}}{N}, \qquad \sigma^2 P_i = \frac{\sigma^2 B + \sigma^2 W_{\text{extra}}}{N^2}$$

where W_{extra} is the additional work inside the body of the test or in the second test. In an indirect test, $\hat{P}_i$ is a function of several measurements.

$$\hat{P}_i = f(\hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n).$$

The normalized 90 percent confidence intervals are given by the expression

$$\left[-\frac{t_{0.95}}{\hat{P}_i} \left[\frac{\sigma^2 P_i}{N_{\text{repeat}}} \right]^{1/2}, \frac{t_{0.95}}{\hat{P}_i} \left[\frac{\sigma^2 P_i}{N_{\text{repeat}}} \right]^{1/2} \right] \tag{6}$$

where $t_{0.95}$ corresponds to the 95 percent percentile of the student's t distribution. Looking at (3)-(6) we see that by increasing N, N_{limit}, and N_{repeat}, we can reduce the variance in our measurements.

D. The Effect of N_{limit} and N_{repeat} on the Variance

One question we have not answered is what should be the magnitude of N_{limit} and N_{repeat} to obtain measurements which give a small σ/μ ratio. These parameters are system dependent and are mainly affected by the resolution of the clock, the concurrent activity on the system, and the particular parameter being measured. We ran several experiments using different values for N_{repeat} and N_{limit} in several machines. Fig. 3 shows the normalized confidence interval of ten parameters for values of N_{limit} such that the each test is run for at least 0.1, 0.2, 0.5, 1.0, 2.0, and 4.0 s on a VAX-11/780. We also obtained measurements for N_{repeat} equal to 5, 10, and 20 observations.

We see that for a fixed value of N_{repeat} the width of the confidence interval of our measurements decreases as the time for the test increases, but for small values of N_{repeat}, there is a limit to how much we can decrease the confidence interval by only increasing the time of the test (N_{limit}). The reason for this is that by increasing the length of the test we reduce the variability due to short-term variations in the concurrent activity of the system. However, the probability of a change in the overall concurrent activity of the system increases with a larger test. This change may produce a greater variance if the size of the sample statistic is small. We see that the best results are obtained for 20 observations and 1-2 s for the duration of the test. In machines with good clock resolution, acceptable results are obtained with 10 observations and 0.2 s for each test.

At one point, we considered not using the mean value measured, as described above, but rather the minimum value, since the minimum should reflect a measurement free of any outside interference. This was found to yield significantly worse predictions; the reason was that "real" runs of the programs in question also suffered the same interference as the measurement program.

VI. Measurements and Some Results

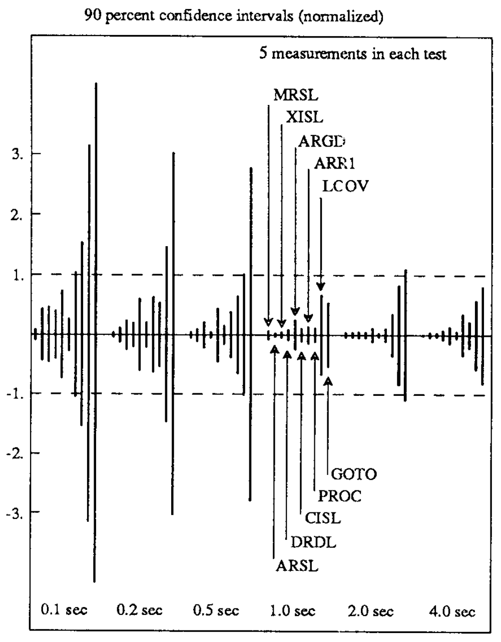
We have run the system characterizer on the machines shown in Table IV, among others. Of the 15 systems shown, four are supercomputers, each implementing single precision floating point with 64 bits. On the other systems, single precision variables are allocated using 32 bits. We gathered two sets of measurements for the Sun 3/260, one using the 68881 coprocessor to execute floating point arithmetic, and another emulating the same functions in software. We also measured the effect of using different Fortran compilers, the VMS FORT compiler and the UNIX BSD f77, both running on the VAX-11/785, in both cases with Ultrix as the operating system. By using the characterizer, we can quantify how much each parameter is affected by the addition of a new hardware feature or by changing the compiler.

The measurements of all parameters are presented in the Appendix in Tables IX-XIII. The parameters are grouped according to Tables II and III, with all magnitudes in units of nanoseconds. Entries with magnitude "< 1" represent parameters that were not detected by the characterizer. This happens when the execution time of the parameter is so small that most of its the execution overlapped with other operations; the total execution time of the program does not depend significantly on the occurrence of these parameters.

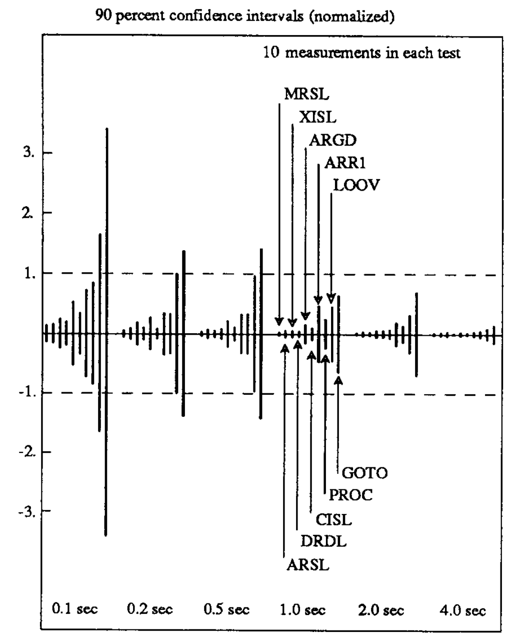
We can see some characteristics of the machines by looking at the results. For example, it is clear from the tables that the performance of the four supercomputers on the execution of double precision arithmetic is significantly lower than that for single precision. (Double precision on those machines, however, is actually quadruple precision for the smaller machines.) Single precision arithmetic operations and intrinsic functions take one order of magnitude less time to execute on these machines than double precision. The greatest difference occurs on the IBM 3090/200 with double precision division. This operation takes almost 700 ns using 64-bit operands, while the same operation with 128-bit operands takes around 75 500 ns. In contrast, the same operation takes less than 8000 ns in any of the three CRAYs.

By looking at the results of the Sun 4/260 and Sun 3/260 (f), we can see the main differences between them. The greatest performance gap is found in floating point (real and complex) arithmetic, intrinsic functions, procedure calls, and parameter passing. For integer arithmetic this difference is smaller.

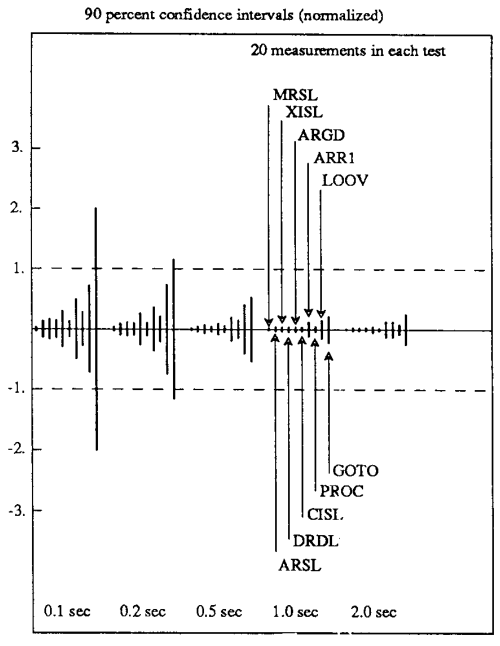
It is also possible to compare our results to published instruction times. However, this is difficult given that our parameters may not map directly to a particular sequence of instructions and that there are many factors affecting the execution times of instructions. For example, on the 68020 the effective address calculation can take from zero to 24 cycles depending on the addressing mode and whether a prefetch instruction or/and an operand read is needed [30], [31]. Nevertheless, Table V shows timing estimates for four intrinsic functions (single precision) and also for the sequence of instructions implementing a procedure call for the 68020. Included in the table are the measurements obtained with the system characterizer. For the instrinsic functions we assumed that the cycle time was 50 ns (20 MHz), and for procedure call 40 ns (25 MHz). These are the clock rates of the MC68881 and MC68020, respectively. We see that except for the logarithm function, our measurements are sufficiently close to the timing estimates. The large difference for logarithm is



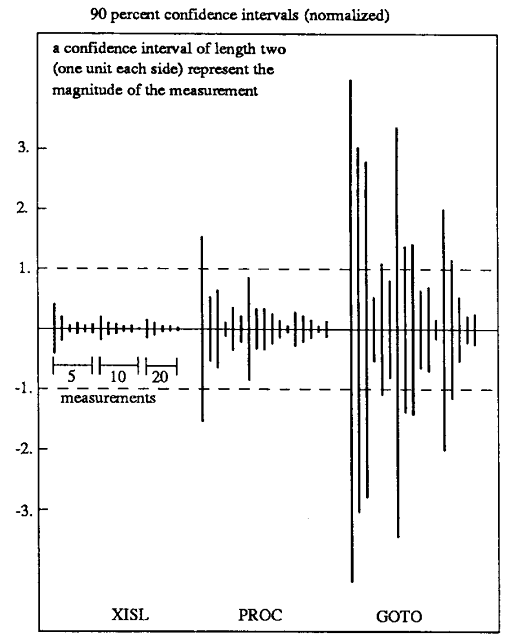
(a)



(b)



(c)



(d)

Fig. 3. Normalized confidence intervals for ten different parameters. In (a), (b), and (c) we show how the length of the test (N_{limit}) and the number of observations (N_{repeat}) affect the confidence interval of the measurements, taken on a VAX-11/780. For a fixed number of observations, an increase in the execution time of the test tends to reduce the length of the confidence interval. (d) shows for three parameters all their confidence intervals plotted together. All confidence intervals are normalized with respect to parameter P_i.

TABLE IV

Characteristics of the Machines. The Size of the Data Type Implementations are in Number of Bits. Sun 3/260 (f) uses the 68881 as a Coprocessor Running at 20 MHz, While the CPU Executes at 25 MHz For the VAX-11/785 We Used Two Fortran Compilers, the VAX FORT 4.7 and the Berkeley f77 1.1.

Characteristics of the machines							
Machine	Name/Location	Operating System	Compiler version	Memory	Integer	Real	
					single	single	double
CRAY Y-MP/832	reynolds.arc.nasa.gov	UNICOS 4.0.8	CFT77 3.0	32 Mw	46	64	128
CRAY-2	navier.arc.nasa.gov	UNICOS 4.0.6	CFT77 3.0	128 Mw	46	64	128
CRAY X-MP/48	NASA Ames	COS 1.16	CFT 1.14	8 Mw	46	64	128
IBM 3090/200	cmsa.berkeley.edu	VM/CMS r.4	FORTRAN v2.3	32 MB	32	64	128
MIPS/1000	cassatt.berkeley.edu	UMIPS-BSD 2.1	F77 v1.21	16 MB	32	32	64
Sun 4/260	rosemary.berkeley.edu	SunOS r.4.0	F77	32 MB	32	32	64
VAX 8600	vangogh.berkeley.edu	UNIX 4.3 BSD	F77 v1.1	28 MB	32	32	64
VAX 3200	atlas.berkeley.edu	Ultrix 2.3	F77 v1.1	8 MB	32	32	64
VAX-11/785 (fort)	pioneer.arc.nasa.gov	Ultrix 3.0	Fort v4.7	16 MB	32	32	64
VAX-11/785 (f77)	pioneer.arc.nasa.gov	Ultrix 3.0	F77 v1.1	16 MB	32	32	64
VAX-11/780	wilbur.arc.nasa.gov	UNIX 4.3 BSD	F77 v2	4 MB	32	32	64
Sun 3/260 (f)	picasso.arc.nasa.gov	UNIX 4.2 r.3.2	F77 v1	16 MB	32	32	64
Sun 3/260	picasso.arc.nasa.gov	UNIX 4.2 r.3.2	F77 v1	16 MB	32	32	64
Sun 3/50	baal.berkeley.edu	UNIX 4.2 r.3.2	F77 v1	4 MB	32	32	64
IBM RT-PC/125	loki.berkeley.edu	ACIS 4.3	F77 v1	4 MB	32	32	64

TABLE V

Execution Estimates Versus Characterization Results

	units	LOGS	EXPS	SINS	TANS	PROC
Timing Est.	cycles	672	598	482	574	113
	nsec	33600	29900	24100	28700	4420
Measurement	nsec	43799	28548	25790	31478	5034
Error		30.5%	4.5%	7.0%	9.7%	13.9%

easily explained by looking at the code generated by the compiler; several additional instructions are included to determine, at execution time, whether to compute $\log(x)$ or $\log(x + 1)$. Therefore, our measurement includes the extra work done.

The effect of different compilers can be seen in the results for the VAX-11/785. The FORT compiler produces code that is significantly faster for complex arithmetic and intrinsic functions, especially single precision intrinsics. There are some surprising results in the case of the exponential operator. While the F77 code is between 2 and 5 times faster using a real base and an integer exponent, the FORT compiler is more than 4 times faster in the case of a real base and a real exponent. A similar situation occurs when the base is integer.

The fact that procedure calls are expensive operations on the VAX architecture can be corroborated when we compare the time it takes to execute this instruction on the VAX 8600 against either the MIPS/1000 or the Sun 4/260. A procedure call is approximately six times slower on the VAX 8600. This large gap is also found for the other VAX implementations when we make the comparison against the Sun 3 or IBM RT-PC. This agrees with previous studies done on the VAX-11/780 that found that procedure calls take on the average 45.25 cycles to execute, while the average VAX instruction takes only 10.6 cycles [7], [17]. On their workload, 14 percent of the time was spent executing this type of instruction.[2]

A. A Reduced Representation of the Performance Measurements

The measurements obtained with the system characterizer make it possible to compare different machine architectures either at the level of the parameters or by predicting the execution times of a set of programs using their parametric dynamic distributions. Predicting the execution time of a program is equivalent to reducing the set of basic measurements to a single number (the execution time) with the dynamic distribution acting as a weighting function. These two types of comparisons represent different extremes. On one side, we have too much information with the raw measurements; it is difficult to identify those parameters that most affect performance without making reference to some particular workload. On the other extreme, a single number representing the executing time gives an illusion of precision by hiding the multidimensional aspects of program execution.

Therefore, it is convenient to represent the parameters in some "reduced" form, in which overall performance is represented using a small number of dimensions, each associated with different aspects of the computation. In this way, it is not only possible to compare the performance of a single operation or the overall performance with respect to a given workload, but also to focus on some particular mode of execution.

B. Combining Measurements and Selecting Weights

The two major issues when we reduce a large number of parameters into a smaller set are how to group the basic measurements, and how much weight to assign to each element.

For the first part we identified a small number of performance "dimensions," each representing either a hardware or a software feature. These "dimensions" should be as independent of each other as possible, and should reflect distinct components of the machine. We use hardware, software, and hybrid parameters. Integer addition is representative of the first group, trigonometric functions of the second, and floating point arithmetic, which in some machines is executed using special hardware and on others by software routines, belongs to the hybrid group.

The second issue, assigning weights to basic parameters, is a more difficult task, given that the impact of a parameter in the performance of a system is a function of the workload. However, this workload dependency is not as serious a

[2] On the more recent VAX 8800 series, procedure calls and returns take only 27.8 cycles, while the average instruction requires 8.8 cycles. Even with this improvement, the VAX 8800 spends 13.8 percent of the time executing procedure calls. For some procedure intensive programs, this number can be as high as 62 percent [8], [9].

TABLE VI
THE 17 REDUCED PARAMETERS, SHOWING BASIC MEASUREMENTS AND THEIR RESPECTIVE WEIGHTS. WITH THE EXCEPTION OF MEMORY BANDWIDTH, THE SUM OF WEIGHTS FOR EACH REDUCED PARAMETER EQUALS ONE. THE SUM OF MEMORY TRANSFER WEIGHTS EQUALS 0.5, BECAUSE THE OPERATION INVOLVES LOADING FROM MEMORY AND WRITING THE RESULTS

Reduced Parameters

1 memory bandwidth (single)				**2 memory bandwidth (double)**			
TRSL	.125	TISL	.125	TCSL	.125	TRDL	.125
TRSG	.125	TISG	.125	TCSG	.125	TRDG	.125
3 integer addition				**6 floating point addition**			
AISL	.500	AISG	.500	ARSL	.500	ARSG	.500
4 integer multiplication				**7 floating point multiplication**			
MISL	.500	MISG	.500	MRSL	.500	MRSG	.500
5 integer arithmetic				**8 floating point arithmetic**			
DISL	.400	DISG	.400	DRSL	.400	DRSG	.400
EISL	.090	EISG	.090	ERSL	.090	ERSG	.090
XISL	.010	XISG	.010	XRSL	.010	XRSG	.010
9 complex precision arithmetic				**10 double arithmetic**			
ACSL	.325	ACSG	.325	ARDL	.325	ARDG	.325
MCSL	.125	MCSG	.125	MRDL	.125	MRDG	.125
DCSL	.040	DCSG	.040	DRDL	.040	DRDG	.040
ECSL	.008	ECSG	.008	ERDL	.008	ERDG	.008
XCSL	.002	XCSG	.002	XRDL	.002	XRDG	.002
11 intrinsic functions (single)				**12 intrinsic functions (double)**			
LOGS	.166	TANS	.166	LOGC	.100	LOGD	.100
EXPS	.166	SQRS	.166	EXPC	.100	EXPD	.100
SINS	.166	MODS	.166	SINC	.100	SIND	.100
				SQRC	.100	SQRD	.100
				TAND	.100	MODD	.100
13 logical operations				**14 pipelining**			
ANDL	.250	CISL	.250	GOTO	.900	GCOM	.100
CRSL	.250	CDRL	.125	**15 procedure calls**			
CCSL	.125			CALL	.750	ARGU	.250
16 address computation				**17 iteration**			
ARR1	.600	ARR3	.100	LOIN	.060	LOIX	.030
ARR2	.300			LOOV	.605	LOOX	.305

problem as in the case of reducing all parameters to a single number. The relative proportion of integer and floating point operations varies greatly from one program to another, but if we focus only on floating point, our experience is that the relative distribution of these operations does not show the same degree of variability. We selected the weights based on extensive statistics of Fortran programs reported in the literature complemented with other statistics produced with our program analyzer [24], [34], [40].

In Table VI, we present the set of parameters and weights that form each of the 17 reduced parameters. Parameters characterizing hardware functional units are: integer addition and multiplication, logical operations, procedure calls, looping, and memory bandwidth (single and double precision). Software characteristics are represented by trigonometric functions (single and double precision). Floating point, double precision and complex arithmetic, pipelining, and address computation belong to the hybrid class.

C. Reduced Measurements and Kiviat Graphs

Fig. 4 shows the values of the reduced parameters, displayed as a Kiviat graph, normalized in each case to the corresponding reduced parameter for the VAX-11/780. Numerical values for the reduced parameters (absolute and normalized) appear in [35], but are omitted here due to lack of space. A Kiviat graph forces the performance graph of the VAX-11/780 to be a circle. Each graph is logarithmic, with each circle representing a change of one order of magnitude with respect to its nearest neighbor. The smallest circle corresponds to a performance equal to one tenth of a VAX-11/780.

From the reduced results, we can identify several differences and similarities between the machines. The memory bandwidth results indicate that the only machines that show the same performance in single and double precision memory bandwidth are the CRAY Y-MP, the CRAY-2, and the CRAY X-MP. Although the single precision memory bandwidth in the IBM 3090 is faster than the CRAY Y-MP (34 ns versus 45 ns), for double precision this situation is reversed (63 ns versus 40 ns).[3] The memory bandwidth reported here does not necessarily match the numbers given by the manufacturer. Our measurements characterize the execution time of a memory transfer assignment in a Fortran program, and for an arbitrary system this transfer is affected by the availability of registers, data cache, write buffer, and other circuitry that improves the data transfer between the CPU and memory.

We can see the effect of different compilers on the VAX-11/785. The difference in performance between the code produced by the two compilers is less than 10 percent in the cases of memory bandwidth with single precision, integer arithmetic, and DO loops. The FORT compiler code is 30 percent faster for real multiplication, 90 percent for complex arithmetic, 130 percent for real division and exponentials, and more than 3 times faster in intrinsic functions. On the other hand, the F77 compiler code is less than 15 percent faster for integer division and address computation. For intensive floating point programs, the code of the FORT compiler clearly outperforms the F77 compiler.

The effect of the floating point coprocessor in the Sun 3/260 is also clear by looking at the results. Using the 68881 increases the performances of intrinsic functions by a factor of more than 13, and for floating point arithmetic by a factor from 2 to 5.

The CRAY Y-MP/832 has the fastest times for floating point and complex arithmetic operations, function calls, array references, branching, and single precision intrinsic functions. In particular, access to array elements is almost 6 times faster in the CRAY than in the IBM 3090/200 and 127 times faster than in the VAX-11/780. The CRAY machines are highly optimized for those operations that are extensively used in scientific programs. The IBM 3090 is the fastest machine for double precision trigonometric functions, single precision memory bandwidth, and logical operations. The CRAY-2 shows performance similar to the CRAY X-MP and Y-MP in integer arithmetic (except integer addition), complex arithmetic, and procedure calls, but memory bandwidth shows a larger difference in both single and double precision.

A comparison between the MIPS/1000 and the Sun-4 shows better performance for the Sun-4 only in memory bandwidth and address computation, and the difference in all cases is less than 15 percent. The MIPS/1000 has an advantage of more than 75 percent in integer multiplication and arithmetic, floating point and complex arithmetic, and intrinsic functions.

[3] The difference between the memory bandwidth measured for the CRAY Y-MP/832 between single and double precision is a result of the measuring tools and the small execution times.

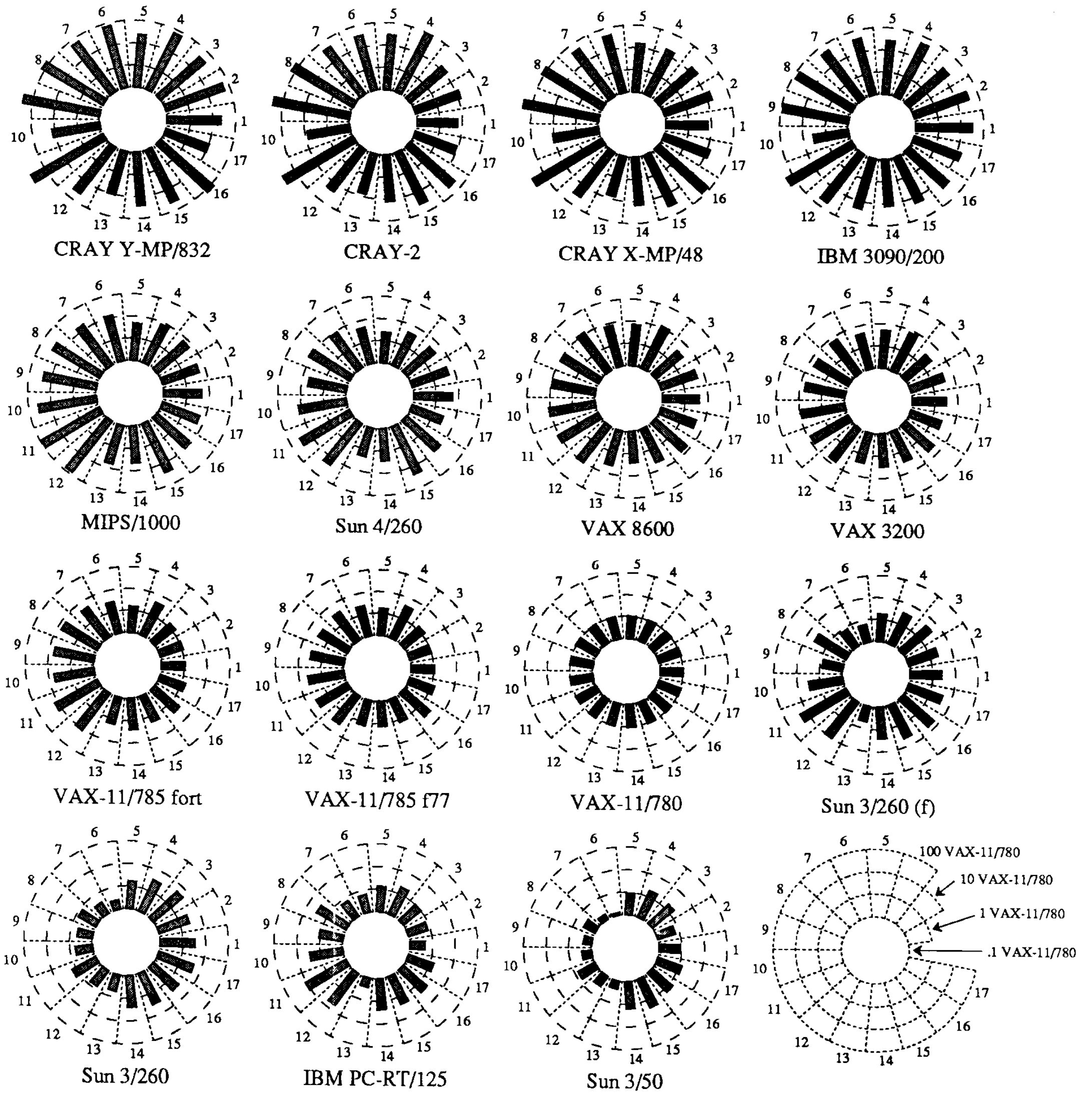


Fig. 4. Performance of the reduced parameters with respect to the VAX-11/780. The concentric circles represents 0.1, 1, 10, and 100 times faster. The closest a performance shape (pershape) is to a circle, the closest the machine is to a VAX-11/780 in terms of how both machines distributed their performance along different computational modes.

Floating point performance on the IBM RT-PC and Sun-3 (50 and 260) is slow compared to other machines and although a coprocessor provides a significant improvement, their performance does not match the performance of comparable minicomputers. For example, the IBM 3090/200 is less than 12 times faster than Sun 3/50 (15 MHz) in integer addition, but 60 times faster than the Sun 3/260 (25 MHz) with 68881 (20 MHz) in floating point addition. The Sun 3/260 is between 4–6 times faster than the VAX-11/780 on procedure calls and array references, but the VAX-11/780 outperforms the Sun 3/260 on single precision floating point addition and multiplication.

VII. Similar Performance Distributions (Performance Shapes)

Consider two machines M_X and M_Y that are identical except for the clock rates. These machines have the property that for any benchmark A their performance ratio (the execution time on one machine divided by the execution time on the other machine) is always a constant; thus, only one benchmark is sufficient to evaluate one against the other. For two arbitrary machines, however, this performance ratio can vary significantly for different benchmarks; it is possible to obtain a wide variety of performance ratios by running a sufficient set of benchmarks. Therefore, it is important to quantify how different is the performance distribution of an arbitrary pair of machines and in this way determine how large we can expect the variability in the performance ratio to be when running a large sample of programs. This metric should group machines according to their performance "shapes" and not by the magnitude of their performance parameters. A *performance shape* (pershape) is the Kiviat graph representing how performance is distributed along the different computational modes (reduced parameters). A pershape tells us not how large a parameter or set of parameters is with respect to other machines but how different machines distribute their performance. In the next subsection, we present a metric that measures how similar are the absolute and normalized pershapes of two arbitrary machines.

A. A Metric for Performance Shapes

We would like a metric that captures the notion of similarity explained in the previous paragraph. By looking at Fig. 4, we clearly see that the pershape of the CRAY Y-MP/832 is very different than the pershapes of the VAX-11/780 or the Sun 3/50. But if we compare the CRAY Y-MP to the CRAY X-MP, or the VAX 8600 to the VAX 3200, we find that except for their relative sizes, the graphs are very similar. It is this informal notion of similarity that we try to capture with the pershape distance.

First, there are several properties that we like our metric to satisfy in addition to the obvious properties required for distances. The pershape distance must be greater or equal to zero, and the distance from any machine to itself must be zero. It should satisfy the triangle inequality. It should be symmetric; the distance from A to B must be equal to the distance from B to A. One essential property for our metric is that if the performance of one machine is increased or decreased by the same quantity in all the dimensions, the new distance does not change. By allowing two different pershapes vectors to have distance zero, we make the pershape distance a semimetric[4] [18]. Every parameter should have the same weight and any arbitrary permutation of the dimensions in both machines should not affect the distance. This means that our metric should be a function only of the relative performance of the machines and not of how we plot them. Making each dimension equally important tends to make the distance workload independent. The last property that we require is that if the performance in one dimension is changed in both machines by the same factor, their relative distance should not be affected.

The following discussion will give the rationale for allowing different pershapes vectors to have distance zero. It is important to understand that we are not trying to measure the difference in performance between two machines, but something completely different. We are interested in the variability of their expected performance. How fast one machine is compared to the other is always a function of the workload we use to evaluate them. What the pershape distance tries to measure is how large is the spectrum of possible comparative performance results when we use any possible workload composition. Therefore, given that two machines have a distance d, if in one machine we increase the performance of every dimension by the same factor (λ), the distance should not be affected. Obviously, the machine will be faster or slower depending on whether λ is greater or less than 1, but the distribution of its performance remains the same. Therefore, its distance to any possible machine should not change. A similar situation happens when we add a constant to a random variable; the mean is affected, but the variance does not change.

[4] In some textbooks, this is called a pseudometric [23], [4]. We will not use the prefix "semi" or "pseudo" and simply refer to it as a metric.

Formally, let $X = \langle x_1, x_2, \cdots, x_n \rangle$ and $Y = \langle y_1, y_2, \cdots, y_n \rangle$ be two performance vectors in $(0, \infty)^n$ representing the pershapes of machines M_X and M_Y. The metric

$$d(X, Y) = \left[\frac{1}{n-1} \sum_{i=1}^{n} \left[\log\left(\frac{x_i}{y_i}\right) - \frac{1}{n} \sum_{j=1}^{n} \log\left(\frac{x_j}{y_j}\right) \right]^2 \right]^{1/2} \tag{8}$$

satisfies the following set of axioms:

i) $d(X, Y) \geq 0$

ii) $d(X, Y) = 0 \quad$ iff $X = \lambda Y$ and $\lambda > 0$

iii) $d(X, Y) = d(Y, X)$

iv) $d(X, Y) \leq d(X, Z) + d(Z, Y)$

v) $d(X_\sigma, Y_\sigma) = d(X, Y) \quad$ for any permutation σ

vi) $d(\langle \lambda x_1, x_2, \cdots, x_n \rangle, \langle \lambda y_1, y_2, \cdots, y_n \rangle) = d(X, Y)$.

Note that (8) is not the only possible distance satisfying the axioms; there are an infinity of different distance metrics with the same basic properties. The only metric property which provides any difficulty to verify is the triangle inequality. To verify axiom iv) we first rewrite (8) as follows:

$$d(X, Y) = \left[\sum_{i=1}^{n} \left[\frac{1}{(n-1)^{1/2}} \left[\log(x_i) - \frac{1}{n} \sum_{j=1}^{n} \log(x_j) \right] - \frac{1}{(n-1)^{1/2}} \left[\log(y_i) - \frac{1}{n} \sum_{j=1}^{n} \log(y_j) \right] \right]^2 \right]^{1/2} \tag{9}$$

then consider the mapping $\phi\colon (0, \infty)^n \to R^n$ defined by

$$\phi(x_i) = \frac{1}{(n-1)^{1/2}} \left[\log(x_i) - \frac{1}{n} \sum_{j=1}^{n} \log(x_j) \right].$$

Now, if we replace $\phi(X)$ and $\phi(Y)$ in (9)

$$d(X, Y) = \left[\sum_{i=1}^{n} (\phi(x_i) - \phi(y_i))^2 \right]^{1/2} \tag{10}$$

we obtain the Euclidean metric for R^n, and the verification of the triangle inequality follows directly from the Cauchy-Schwarz inequality [18].

In our presentation of the pershape distance, we did not specify whether vectors X and Y represent absolute or normalized pershapes. Computing a function on a normalized set of values does not always preserve some elementary proper-

TABLE VII
PAIRS OF MACHINES WITH SMALLEST AND LARGEST PERSHAPE DISTANCE

Most Similar Machines

	machine	machine	distance
001	VAX 8600	VAX 3200	0.187
002	VAX 8600	VAX-11/785 (f77)	0.214
003	VAX 3200	VAX-11/785 (f77)	0.235
004	Sun 3/50	Sun 3/260	0.291
005	VAX 3200	VAX-11/780	0.425
006	VAX-11/785 (f77)	VAX-11/785 (fort)	0.432
007	CRAY Y-MP/832	CRAY X-MP/48	0.454
008	MIPS/1000	VAX-11/785 (fort)	0.478
009	MIPS/1000	Sun 4/260	0.493
010	VAX 8600	VAX-11/780	0.498
011	VAX 3200	VAX-11/785 (fort)	0.509
012	VAX 8600	VAX-11/785 (fort)	0.516
013	CRAY-2	CRAY X-MP/48	0.518
014	VAX-11/785 (f77)	VAX-11/780	0.519
015	IBM RT-PC/125	Sun 3/260 (f)	0.522
016	CRAY Y-MP/832	CRAY-2	0.532
017	CRAY Y-MP/832	IBM 3090/200	0.661
018	Sun 4/260	VAX-11/785 (fort)	0.663
019	VAX-11/785 (fort)	IBM RT-PC/125	0.672
020	Sun 4/260	IBM RT-PC/125	0.684

Least Similar Machines

	machine	machine	distance
105	CRAY-2	Sun 3/260	1.753
104	CRAY X-MP/48	Sun 3/260	1.725
103	MIPS/1000	Sun 3/260	1.661
102	CRAY X-MP/48	Sun 3/50	1.648
101	CRAY-2	Sun 3/50	1.647
100	MIPS/1000	Sun 3/50	1.591
099	CRAY Y-MP/832	Sun 3/260	1.562
098	VAX-11/785 (fort)	Sun 3/260	1.523
097	CRAY Y-MP/832	Sun 3/50	1.503
096	VAX-11/785 (fort)	Sun 3/50	1.445
095	IBM 3090/200	Sun 3/260	1.434
094	IBM 3090/200	Sun 3/50	1.421
093	CRAY-2	Sun 3/260 (f)	1.420
092	Sun 4/260	Sun 3/260	1.345
091	CRAY X-MP/48	Sun 3/260 (f)	1.303
090	VAX-11/785 (f77)	Sun 3/260	1.300
089	VAX 8600	Sun 3/260	1.296
088	Sun 4/260	Sun 3/50	1.286
087	CRAY-2	VAX-11/780	1.264
086	VAX 8600	Sun 3/50	1.250

ties. The output of the function may change when we normalize the inputs. It is important to see how our metric behaves when we normalize the set of reduced parameters.

Let X be an absolute pershape vector and X_Z be a normalized vector obtained by dividing each component x_i of X with the corresponding element in Z

$$X_Z = \left\langle \frac{x_1}{z_1}, \frac{x_2}{z_2}, \cdots, \frac{x_n}{z_n} \right\rangle .$$

In linear algebra terms, normalizing vector X with respect to vector Z means applying a linear transformation T to vector X, such that the transformation matrix associated with T is diagonal. The matrix is zero everywhere except in the diagonal, with $1/z_i$ as the diagonal element i. Now the normalized distance is given by

$$d(TX, TY) = d(X_Z, Y_Z) = d\left(\left\langle \frac{x_1}{z_1}, \cdots, \frac{x_n}{z_n} \right\rangle, \left\langle \frac{y_1}{z_1}, \cdots, \frac{y_n}{z_n} \right\rangle\right) = d(X, Y).$$

If we substitute the normalized parameters in (8), we see that the distance does not change. It is also easy to see that this property is enforced by axioms v) and vi). Thus, we say that distance $d(X, Y)$ is isometric with respect to diagonal linear transformations.

In addition to measuring the distance between two performance vectors, the metric also gives information on which parameters will most affect the benchmark results between two machines. By ordering the terms inside of the first summation in (8), we find that the largest terms will be the ones that will contribute more to the summation, and therefore to the distance.

1) Similarity Results: Pershape distances were computed for all pairs of machines to detect which were the most and least similar machines. The most and least similar 20 are reported in Table VII. The table shows that the most similar machines are the VAX 8600, VAX 3200, and the VAX-11/785,

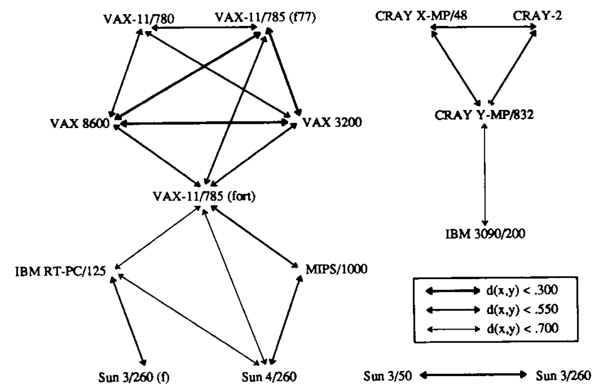


Fig. 5. All machines with performance distance less than 0.700 are joined by a double arrow. The pershape distance identifies clusters of machines with similar performance distributions.

all using the F77 compiler. Other machines that are also close to each other are the Sun 3/50 and the Sun 3/260, both running without the 68881. The differences between these two machines are the clock, the cache, and the memory. The Sun 3/50 runs at 15 MHz, does not have a cache, and uses standard memory chips. The Sun 3/260 runs at 25 MHz, has 64 kbytes of virtual address write-back cache, and uses ECC for memory.

It is possible to use the results in Table VII to identify not only pairs of machines with similar pershapes, but also clusters of machines. Fig. 5 illustrates one possible diagram showing for all the machines a bidirectional arrow joining the machines that have a distance less than 0.7. Different arrows are used to show how close the machines are. In the diagram, we see three connected components, one formed by the supercomputers, another by the small workstations without floating point coprocessors, and a large component mainly formed by two groups having a common neighbor. The closest of the two groups is formed by the machines implementing the VAX architecture and using the F77 compiler. The other group is formed by fast workstations. The VAX-11/785 using the FORT compiler acts a bridge between the two groups.

TABLE VIII

EXECUTION RATIOS BETWEEN PAIR OF MACHINE AND COMPARISON AGAINST THEIR PERFORMANCE DISTANCES. THE COLUMNS ON THE LEFT SHOW THE EXECUTION TIMES (IN SECONDS). THE UPPER RIGHT COLUMNS GIVE THE EXECUTION RATIOS. THE MAXIMUM AND MINIMUM ENTRIES CORRESPOND TO THE RATIO OF LARGEST AND SMALLEST EXECUTION RATIOS. MACHINES WITH A SMALL PERFORMANCE DISTANCE HAVE LESS VARIABILITY IN THEIR RELATIVE SPEED AND THIS SHOULD CORRESPOND TO A SMALL MAX/MIN VALUE. WE GIVE RESULTS FOR THE SUN 3/260 WITH COPROCESSOR ((f)I), AND WITHOUT IT (II).

	Sun 3/260		IBM RT-PC	Sun 3/50	execution ratios					
program	(f) I	II	III	IV	II/I	III/I	IV/I	III/II	IV/II	IV/III
Alamos	1547.9 s	2838.9 s	3881.9 s	6273.2 s	1.83	2.51	4.05	1.37	2.21	1.62
Baskett	3.92 s	3.88 s	6.20 s	7.06 s	0.99	1.58	1.80	1.60	1.82	1.14
Erathostenes	0.64 s	0.64 s	1.10 s	0.90 s	1.00	1.72	1.59	1.72	1.59	0.93
Linpack	184.9 s	338.5 s	473.9 s	763.7 s	1.83	2.56	4.13	1.40	2.26	1.61
Livermore	507.1 s	1103.1 s	1610.1 s	2457.0 s	2.18	3.18	4.85	1.46	2.23	1.53
Mandelbrot	41.88 s	75.88 s	105.43 s	163.94 s	1.81	2.52	3.92	1.39	2.16	1.56
Shell	1.68 s	1.72 s	4.68 s	3.14 s	1.02	2.79	1.87	2.72	1.83	0.67
Smith	338.3 s	406.7 s	545.10 s	914.8 s	1.20	1.61	2.70	1.34	2.25	1.68
Whetstone	4.74 s	15.28 s	12.05 s	34.24 s	3.22	2.54	7.22	0.79	2.24	2.84
				minimum	0.99	1.58	1.59	0.79	1.59	0.67
				geom. mean	1.55	2.27	3.18	1.46	2.05	1.40
				maximum	3.22	3.18	7.22	2.72	2.26	2.84
				max/min	3.26	2.01	4.53	3.46	1.41	4.23
				d(x,y)	0.96	0.52	0.93	1.23	0.29	1.13

B. An Application of Pershape Distances

By using (8), it is possible not only to compute the distance between two machines but also to quantify which "composite" parameters contribute most to unbalance the overall performance ratio between the two machines. In Table VIII, the execution times of nine programs are given for four of the machines. The table also includes the performance ratio between them, the maximum, minimum, and geometric mean of their performance ratios, the maximum ratio of their relative performance, and their pershape distance.

The programs used as benchmarks have different execution distributions and can be grouped in the following way: Shell, Erathostenes, and Baskett are integer programs; Alamos [19], [37], Linpack [14], [16], Livermore [28], and Mandelbrot are floating point intensive programs; Whetstone [12] is a floating point and intrinsic function program; and the Smith benchmark [38] mixes floating point, integer, and logical operations. Baskett also executes a large proportion of function calls.

The results in the table show the relation between the pershape distance and the range of possible benchmark results we can obtain when running a group of benchmarks. The pershape distance between the Sun 3/260 (without 68881) and the Sun 3/50 is only 0.29 and the interval of benchmark results is just 1.41. The difference between the smallest ratio (1.59) and the largest (2.25) is 41 percent. The same small distance is found between the IBM RT-PC and the Sun 3/260 (which uses a coprocessor). Machines with large distance pershapes give a large variation in the benchmark results, but the relation is not as clear as in the other cases. A possible explanation is that our program sample is not large enough, and certain types of operations that contribute to a large distance are not present in a large enough proportion to skew the benchmark results. The results do show that the Sun 3/50 can be 1.6 times slower than the Sun 3/260 (with 68881) in a predominantly integer benchmark, but 7.2 times slower in a benchmark with a high number of intrinsic functions. This is consistent with the performance ratios of the parameters representing integer operations and intrinsic functions. By looking at the distances between a group of machines, it is possible to identify which characteristics of the benchmarks will give a more complete evaluation of the systems. In contrast, programs that only exploit one or two characteristics will give skewed results.

VIII. WEAK POINTS IN THE CHARACTERIZER

The latest version of the characterizer incorporates several additional parameters that were previously ignored. This has increased the number of parameters from 76 to 102. Complex variables and a better characterization of intrinsic functions form most of the new parameters. Even in this extended model there are several factors that have not been characterized.

1) Locality and Cache Memory: Code that exhibits different locality than our experiments affects the cache hit ratio and in consequence the access time for data and/or instructions. Measuring how the access time will be affected by different parts of the program will probably not be possible using a machine characterizer. By running some experiments with different degrees of locality we have found a variation of between four to ten percent.

2) Branching: The size of the branch affects the execution time by modifying the locus of execution. If the target of the branch is to a nonresident page this may involve a page fault and a context switch. A context switch normally involves flushing the cache and this forces a "cold" start on cache references.

3) Hardware and/or Software Interlocks: In pipelined machines, the time to produce a new result depends on the context in which the instruction is executed. This normally depends (in addition to the effective execution time) on the functional and data dependencies with respect to previously scheduled instructions. As in the previous two factors, this is difficult to measure from a high-level program.

4) Machine Idioms: Special cases of some instructions are optimized to improve execution time. These idioms are used by the compiler whenever possible. Without knowledge of the architecture and the compiler, it is not possible to detect which are the idioms of a given machine. For example, in machines with autoincrement and autodecrement addressing modes, these modes may be used in statements like $i = i + 1$.

5) "Random" Noise Produced by Concurrent Activity: Although we address this problem in Section V-D, there is still a problem left when we run in a loaded system. A small increase in the load of the system tends to affect the measurements

of some parameters, in particular array address computation, branches, and loop overhead.

6) Optimization: In this study, we only considered unoptimized code; the characterizer was compiled and run with optimization disabled. Even when it is not difficult to detect which transformations an optimizing compiler can apply, it is not clear how we should modify the execution time model to include optimized programs. Parsing the code and detecting which optimizations are possible and deciding for these which ones are going to be applied by a particular compiler seems to require a "super-optimizer." It is outside the scope of this research to write such program; we are working on other techniques to characterize optimization.

IX. Conclusions and Summary

In this paper, we have presented a model for machine characterization based on a large number of high-level parameters representing operations for an abstract Fortran machine. This provides a uniform model in which machines with different architectures can be compared. It is possible to detect differences and similarities between machines with respect to individual parameters. In addition, we have presented a set of composite parameters that provides a more compact way of representing the effect of hardware or software features in the execution time of programs. Based on these composite parameters we presented the concept of performance shape to show how different machines distribute their possible performance in different ways. We defined a metric to measure the similarity between two pershapes and show how this distance can be used to classify machines and the metric's relation to the variation in benchmark results.

Using the characterization results for the reduced parameters, it is possible to make estimates for the execution time of programs and in this way study the sensitivity of the execution time with respect to variations in the workload; this last aspect will be presented in a forthcoming paper [36]. We think that our approach will advance the state of the art of performance evaluation in several ways.

1) A uniform "high-level" model of the performance of computer systems allows us to make better comparisons between different architectures and identify their differences and similarities when the systems execute a common workload.

2) Using the characterization to predict performance provides us with a mechanism to validate our assumptions on how the execution time depends on individual components of the system.

3) With a uniform model that can be used for all machines sharing a common mode of computation, it is possible to define metrics that permit more extensive comparisons and in this way obtain a better understanding of the behavior of each system.

4) We can study the sensitivity of the system to changes in the workload, and in this way detect imbalances in the architectures.

5) The results obtained with the system characterizer give insight into the implementation of the CPU architecture, and the machine designers can use the results to improve future implementations.

6) Application programmers and users can identify the most time consuming parts of their programs and measure the impact of new "improvements" on different systems.

7) For procurement purposes, this is a less expensive and more flexible way of evaluating computer systems and new architectural features. Although the best way to evaluate a system is to run a real workload, a more extensive and intensive evaluation can be made using system characterizers to select a small number of computers for subsequent on-site evaluation.

In the last 30 years, we have seen an explosion of new ideas in many fields of computer science, but one problem that has not received much attention is how to make a fair comparison between two different architectures. Given the impact that computers have in all aspects of society we cannot afford to continue characterizing the performance of such complex systems using MIPS, MFLOPS, or DHRYSTONES as our units of measure.

Appendix

TABLE IX
Characterization Results for Groups 1–3. A Value 1< Indicates that the Parameter Was Not Detected by the Experiment

Group 1: Floating Point Arithmetic Operations (single, local)

machine	SRSL	ARSL	MRSL	DRSL	ERSL	XRSL	TRSL
CRAY Y-MP/832	13	46	111	210	660	4150	96
CRAY-2	39	70	101	250	78	4180	112
CRAY X-MP/432	82	76	154	357	91	5035	281
IBM 3090/200	1<	82	140	684	129	4952	60
MIPS/1000	67	269	437	976	543	53018	499
Sun 4/260	104	755	788	2496	4724	60430	533
VAX 8600	72	425	575	1610	1097	217676	509
VAX 3200	262	805	999	2013	1847	361666	587
VAX-11/785 fort	263	1282	1524	3778	16305	82006	1799
VAX-11/785 f77	246	1371	1924	4034	3740	648082	2065
VAX-11/780	1086	3215	6739	9322	11041	2066420	1598
Sun 3/260 (f)	1978	5543	8709	11394	15998	58901	1293
Sun 3/260	1<	13580	19118	23003	31612	2205175	1286
Sun 3/50	1<	26420	40246	46476	60818	4743815	3076
IBM RT-PC/125	3639	5684	10715	12304	12437	231989	6235

TABLE IX (*Continued*)

Group 2: Floating Point Arithmetic Operations (complex, local)

machine	SCSL	ACSL	MCSL	DCSL	ECSL	XCSL	TCSL
CRAY Y-MP/832	30	85	267	497	818	10466	147
CRAY-2	32	110	221	386	48	17167	199
CRAY X-MP/432	63	124	271	511	1<	13168	319
IBM 3090/200	26	215	679	3218	2940	13912	97
MIPS/1000	121	926	1727	12025	9004	72791	1097
Sun 4/260	1<	8034	11808	29356	7561	130805	663
VAX 8600	275	1438	3523	39419	17876	326399	974
VAX 3200	792	2287	6925	47240	30134	510817	1072
VAX-11/785 fort	531	2653	7542	53236	26842	314924	3514
VAX-11/785 f77	1074	4717	10206	88085	83278	966246	4703
VAX-11/780	1319	9679	38202	328270	170337	3584596	3796
Sun 3/260 (f)	436	27270	83719	353726	133222	446378	1382
Sun 3/260	1<	31812	109547	604151	183495	5417755	1095
Sun 3/50	265	63460	231185	1233098	453373	11405138	8310
IBM RT-PC/125	471	26969	47498	194262	183060	678778	5101

Group 3: Integer Arithmetic Operations (single, local)

machine	SISL	AISL	MISL	DISL	EISL	XISL	TISL
CRAY Y-MP/832	1<	39	106	271	1113	1131	82
CRAY-2	1<	61	62	324	126	131	114
CRAY X-MP/432	1<	91	414	714	396	755	320
IBM 3090/200	1<	76	143	439	163	358	73
MIPS/1000	1<	227	945	2577	1111	2146	475
Sun 4/260	1<	286	1634	3918	5882	7979	219
VAX 8600	1<	357	628	1591	896	1883	462
VAX 3200	1<	490	895	2206	1273	2592	750
VAX-11/785 fort	1<	1002	1615	7292	1760	28928	2259
VAX-11/785 f77	1<	1088	1789	7053	2309	5142	2182
VAX-11/780	1<	1327	6924	10502	7779	15803	2186
Sun 3/260 (f)	1<	298	2212	4011	13979	17174	393
Sun 3/260	1<	237	2280	4119	14708	17398	251
Sun 3/50	1<	813	3898	7039	29262	36348	856
IBM RT-PC/125	1<	1497	3438	8837	4063	7581	2478

TABLE X

Characterization Results for Groups 4–6. A Value 1< Indicates that the Parameter Was Not Detected by the Experiment

Group 4: Floating Point Arithmetic Operations (double, local)

machine	SRDL	ARDL	MRDL	DRDL	ERDL	XRDL	TRDL
CRAY Y-MP/832	2	917	1626	5473	4804	108121	13
CRAY-2	1<	1974	2752	7355	2072	194054	1<
CRAY X-MP/432	69	1122	1812	6392	1073	138645	206
IBM 3090/200	1<	424	964	75656	1493	48282	154
MIPS/1000	117	346	581	1556	838	49780	632
Sun 4/260	290	986	1228	4665	7046	133573	1058
VAX 8600	220	754	1725	5812	2841	208984	876
VAX 3200	276	1367	1896	4063	3256	353750	1178
VAX-11/785 fort	1047	2280	4243	7996	23386	177403	3920
VAX-11/785 f77	929	2893	5460	8921	9517	636736	5637
VAX-11/780	1142	10589	24687	48235	33181	2044644	5483
Sun 3/260 (f)	2159	5819	9272	11942	17793	112601	2610
Sun 3/260	1172	23804	49458	73051	46323	2482220	1364
Sun 3/50	2068	54245	100594	132284	110522	5504681	6682
IBM RT-PC/125	5765	6889	8125	12611	14110	200954	4623

Group 5: Floating Point Arithmetic Operations (single, global)

machine	SRSG	ARSG	MRSG	DRSG	ERSG	XRSG	TRSG
CRAY Y-MP/832	13	46	111	210	660	4150	96
CRAY-2	95	124	216	392	72	3811	506
CRAY X-MP/432	90	73	152	354	83	5052	287
IBM 3090/200	12	80	129	685	152	4901	60
MIPS/1000	70	268	435	973	536	50784	364
Sun 4/260	145	778	855	2573	4739	60387	573
VAX 8600	254	483	598	1600	1040	215039	408
VAX 3200	400	878	1076	2159	1770	361567	554
VAX-11/785 fort	998	1244	1501	3619	16318	81494	1430
VAX-11/785 f77	219	1378	1928	4049	3727	651381	2070
VAX-11/780	1517	3304	6646	9539	10649	2056123	1164
Sun 3/260 (f)	1948	5616	8945	11662	16018	58321	1157
Sun 3/260	1<	13433	18920	22865	31453	2139632	716
Sun 3/50	1<	26943	40586	47035	58663	4671805	5092
IBM RT-PC/125	3156	5629	9684	12287	13054	230580	6636

TABLE X (*Continued*)

Group 6: Floating Point Arithmetic Operations (complex, global)

machine	SCSG	ACSG	MCSG	DCSG	ECSG	XCSG	TCSG
CRAY Y-MP/832	30	85	267	497	818	10466	147
CRAY-2	92	167	303	513	18	16738	512
CRAY X-MP/432	78	117	265	511	1<	13177	335
IBM 3090/200	27	230	682	3162	2983	13965	102
MIPS/1000	121	927	1730	12049	8992	73007	1101
Sun 4/260	63	8027	12078	29703	7573	130146	664
VAX 8600	551	1544	3802	38787	17812	326228	1159
VAX 3200	519	2859	7334	46918	31358	511323	1599
VAX-11/785 fort	1055	3210	8827	52768	24348	298978	4393
VAX-11/785 f77	1144	4695	10039	87745	83649	962812	4680
VAX-11/780	1684	9778	35853	322237	168501	3679780	3297
Sun 3/260 (f)	1<	27975	83103	352636	134477	453818	1519
Sun 3/260	3695	29210	107292	586288	191029	5307737	1<
Sun 3/50	2383	63526	231688	1233524	448297	11359785	8016
IBM RT-PC/125	555	26948	47435	197036	182374	693827	5216

TABLE XI
CHARACTERIZATION RESULTS FOR GROUPS 7–10. A VALUE 1< INDICATES THAT THE PARAMETER WAS NOT DETECTED BY THE EXPERIMENT

Group 7: Integer Arithmetic Operations (single, global)

machine	SISG	AISG	MISG	DISG	EISG	XISG	TISG
CRAY Y-MP/832	1<	39	106	271	1113	1131	82
CRAY-2	1<	161	89	485	144	153	607
CRAY X-MP/432	1<	93	405	716	405	751	327
IBM 3090/200	1<	79	151	439	170	393	82
MIPS/1000	1<	227	942	2580	1110	2143	476
Sun 4/260	1<	421	1728	4022	6022	7972	252
VAX 8600	1<	522	606	1593	990	2010	622
VAX 3200	1<	594	1028	2202	1484	2852	826
VAX-11/785 fort	1<	1113	1888	7428	1857	29838	2269
VAX-11/785 f77	1<	1094	1788	7025	2279	5089	2167
VAX-11/780	1<	1616	7166	10731	8002	16036	2156
Sun 3/260 (f)	1<	438	2116	4015	14156	17771	427
Sun 3/260	1<	381	2121	4050	14212	16824	218
Sun 3/50	1<	937	3537	6887	29760	36609	738
IBM RT-PC/125	1<	1459	3422	8865	3956	7553	2438

Group 8: Floating Point Arithmetic Operations (double, global)

machine	SRDG	ARDG	MRDG	DRDG	ERDG	XRDG	TRDG
CRAY Y-MP/832	2	917	1626	5473	4804	108121	13
CRAY-2	1<	2051	2858	7360	2283	202494	302
CRAY X-MP/432	69	1122	1821	6342	1108	138935	222
IBM 3090/200	1<	421	963	73307	912	41150	154
MIPS/1000	108	349	587	1561	854	58483	679
Sun 4/260	278	961	1167	4586	7078	133796	1060
VAX 8600	252	796	1611	5905	2828	206974	817
VAX 3200	268	1515	2175	4238	3307	353997	1080
VAX-11/785 fort	1207	2202	4106	8044	23236	171685	3882
VAX-11/785 f77	968	2268	4498	7916	9192	636084	4461
VAX-11/780	1274	10848	24648	47719	34214	2024890	4551
Sun 3/260 (f)	2354	5790	9275	11817	18065	112638	2477
Sun 3/260	549	23648	49458	72612	45612	2562978	2664
Sun 3/50	2436	54749	100942	133431	110630	5495105	4046
IBM RT-PC/125	3799	6017	10228	13526	14088	203231	7612

Group 9,10: Conditional and Logical Parameters

machine	ANDL	CRSL	CCSL	CISL	CRDL	ANDG	CRSG	CCSG	CISG	CRDG
CRAY Y-MP/832	14	287	315	282	335	14	287	315	282	335
CRAY-2	36	95	237	96	1873	85	317	430	337	1963
CRAY X-MP/432	45	226	335	229	1243	46	228	322	231	1256
IBM 3090/200	104	94	106	73	214	111	138	171	161	259
MIPS/1000	185	471	375	337	603	183	474	404	335	602
Sun 4/260	310	1217	3767	236	1566	455	1333	4168	655	1585
VAX 8600	304	653	680	464	867	321	868	991	743	1022
VAX 3200	389	1127	1295	767	1603	412	1207	1371	844	1602
VAX-11/785 fort	954	1378	1727	1033	2649	1013	1747	2348	1116	2264
VAX-11/785 f77	769	1937	1966	1467	2578	768	1909	1937	1477	2629
VAX-11/780	1091	2823	2768	1987	3914	1100	3057	3674	2481	4573
Sun 3/260 (f)	414	6814	16257	329	7171	588	7093	15922	741	7057
Sun 3/260	394	5542	10938	332	10730	604	5728	11985	847	9765
Sun 3/50	803	13243	29047	559	22442	1399	14382	27969	1625	25800
IBM RT-PC/125	1005	16166	16033	2012	15919	1029	15430	15700	2130	15993

TABLE XII
CHARACTERIZATION RESULTS FOR GROUPS 11–15. A VALUE 1< INDICATES THAT THE PARAMETER WAS NOT DETECTED BY THE EXPERIMENT

Group 11,12: Function Call, Arguments and References to Array Elements

machine	PROC	ARGU	ARR1	ARR2	ARR3	IADD
CRAY Y-MP/832	512	61	42	49	60	12
CRAY-2	574	40	122	159	200	1<
CRAY X-MP/432	583	73	59	104	148	2
IBM 3090/200	1162	70	128	410	746	17
MIPS/1000	797	139	523	1044	1592	1<
Sun 4/260	918	67	384	1004	1490	13
VAX 8600	4670	610	478	1223	2137	1<
VAX 3200	6991	957	668	1934	3316	1<
VAX-11/785 fort	11678	1515	1320	2897	5578	844
VAX-11/785 f77	16421	1526	995	2701	5057	32
VAX-11/780	19931	1783	2126	9592	18518	1<
Sun 3/260 (f)	5034	397	448	1661	2600	2
Sun 3/260	6548	594	990	3834	3484	1<
Sun 3/50	8838	1535	2042	6396	8759	100
IBM RT-PC/125	9395	991	2212	2406	4536	1<

Group 13,14: Branching and DO loop Parameters

machine	GOTO	GCOM	LOIN	LOOV	LOIX	LOOX
CRAY Y-MP/832	1<	406	1015	315	627	368
CRAY-2	15	692	1263	353	264	513
CRAY X-MP/432	25	483	966	180	1307	293
IBM 3090/200	38	460	660	130	952	353
MIPS/1000	137	1010	1938	417	1643	945
Sun 4/260	302	984	3378	1007	2320	1638
VAX 8600	262	1705	2540	396	6223	1070
VAX 3200	128	2117	3916	975	5336	1634
VAX-11/785 fort	277	1691	13042	972	11747	3124
VAX-11/785 f77	332	4262	8323	1621	7644	2768
VAX-11/780	588	4783	2525	2552	17363	4558
Sun 3/260 (f)	258	1742	2863	567	3256	1509
Sun 3/260	268	1694	1657	524	1957	1411
Sun 3/50	394	3001	6558	1976	5765	3776
IBM RT-PC/125	119	3395	11368	1236	5425	3396

Group 15: Intrinsic Functions (single precision)

machine	EXPS	LOGS	SINS	TANS	SQRS	ABSS	MODS	MAXS
CRAY Y-MP/832	1453	1314	1423	1514	1038	1<	265	177
CRAY-2	1980	1855	2067	2136	266	25	383	328
CRAY X-MP/432	1826	1627	1846	1985	1356	1<	318	200
IBM 3090/200	2893	2887	2805	4119	2534	37	1094	435
MIPS/1000	6612	5680	5751	5156	6745	61	7215	1470
Sun 4/260	13560	14197	12081	20338	14520	450	23141	4758
VAX 8600	67798	52587	42683	70577	23883	1285	26471	3275
VAX 3200	109786	77167	63001	99637	32436	2108	38300	4563
VAX-11/785 fort	27212	28438	39474	70494	22634	215	42421	4101
VAX-11/785 f77	204824	240223	109462	138871	56848	2996	88497	8302
VAX-11/780	690106	765999	468763	857151	177536	4230	186125	12234
Sun 3/260 (f)	43799	28548	25790	31478	12627	464	15571	15528
Sun 3/260	367032	443458	574151	686006	61509	1<	49869	18798
Sun 3/50	770610	950878	1272922	1512997	92447	4700	129932	47730
IBM RT-PC/125	27466	22327	23168	26511	7014	47189	179593	41101

TABLE XIII
CHARACTERIZATION RESULTS FOR GROUPS 16–18. A VALUE 1< INDICATES THAT THE PARAMETER WAS NOT DETECTED BY THE EXPERIMENT

Group 16: Intrinsic Functions (double precision)

machine	EXPD	LOGD	SIND	TAND	SQRD	ABSD	MODD	MAXD
CRAY Y-MP/832	51052	58111	32289	71166	8689	28	9581	2200
CRAY-2	88428	94268	67440	146937	12100	431	19506	1021
CRAY X-MP/432	70511	64914	37931	83390	9751	21	9459	727
IBM 3090/200	20471	21893	19390	28520	10193	70	76571	858
MIPS/1000	8565	7508	7997	7747	9330	39	6985	2385
Sun 4/260	22261	22220	21184	36096	27382	504	18995	6106
VAX 8600	67151	52267	41751	69792	23310	1755	24244	5083
VAX 3200	108029	79491	63237	101020	31103	3001	35793	7175
VAX-11/785 fort	51621	51081	97932	158473	30389	693	71349	7050
VAX-11/785 f77	203491	238536	107896	137069	55608	5906	84380	17867
VAX-11/780	701933	776686	467842	856357	179556	7504	176955	22079
Sun 3/260 (f)	46526	32093	28500	32619	13966	312	17058	19584
Sun 3/260	965293	1096555	1009146	1132512	94349	1<	60492	22428
Sun 3/50	2080362	2332882	2210543	2418819	175124	1<	150656	67713
IBM RT-PC/125	38928	34208	34753	37343	13901	11669	120334	44149

TABLE XIII (*Continued*)

Groups 17,18: Intrinsic Functions (integer and complex)

machine	ABSI	MODI	MAXI	EXPC	LOGC	SINC	SQRC	ABSC
CRAY Y-MP/832	76	563	127	6093	4478	5027	4282	1784
CRAY-2	51	545	202	9299	7827	9244	3761	1618
CRAY X-MP/432	58	1644	192	7913	5755	6553	5020	2309
IBM 3090/200	93	541	399	6948	5384	7081	6188	2302
MIPS/1000	169	2607	1415	21639	20454	25382	17361	6955
Sun 4/260	1027	3261	2957	87132	46483	123282	73181	38489
VAX 8600	1381	2546	2983	168775	145238	262011	87857	47671
VAX 3200	1498	3640	3816	266255	233369	438531	118507	62425
VAX-11/785 fort	563	11022	3367	156792	81107	88997	102146	54562
VAX-11/785 f77	2897	8970	8096	458645	491650	760555	199835	113818
VAX-11/780	4262	16165	10719	1749767	1637191	2510877	626909	299780
Sun 3/260 (f)	1665	3721	3825	178628	196966	231844	232360	26185
Sun 3/260	3186	6529	3414	2722348	2339489	3656209	649596	168115
Sun 3/50	9953	15760	13886	5734016	5100953	7775319	1338978	364009
IBM RT-PC/125	2549	9232	8196	410201	233930	511218	380805	258205

Acknowledgment

We would like to thank K. Stevens, Jr. and E. Barszcz for their comments on earlier drafts and for providing access to facilities at NASA Ames, as well as J. A. Ambros-Ingerson and H. Touati who made many useful suggestions.

References

[1] D. H. Bailey and J. T. Barton, "The NAS kernel benchmark program," NASA Tech. Memo. 86711, Aug. 1985.

[2] D. H. Bailey, "NAS kernel benchmark results," in *Proc. First Int. Conf. Supercomput.*, St. Petersburg, FL, Dec. 16–20, 1985, pp. 341–345.

[3] H. E. Bal and A. S. Tanenbaum, "Language- and machine-independent global optimization on intermediate code," *Comput. Languages*, vol. 11, no. 2, pp. 105–121, 1986.

[4] N. Bourbaki, *Elements of Mathematics: General Topology*. New York: Springer-Verlag, 1989.

[5] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *Proc. Supercomputing '88 Conf.*, Kissimmee, FL, Nov. 1988.

[6] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze, "Toward real-time performance benchmarks for ADA," *Commun. ACM*, vol. 29, no. 8, pp. 760–778, Aug. 1986.

[7] D. W. Clark and H. M. Levy, "Measurement and analysis of instruction set usage in the VAX-11/780," in *Proc. 9th Annu. Symp. Comput. Architecture*, Apr. 1982.

[8] D. W. Clark, "Pipelining and performance in the VAX 8800 processor," in *Proc. Second Int. Conf. Architectural Support Programming Languages Oper. Syst.*, Palo Alto, CA, Oct. 1987, pp. 173–177.

[9] D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring VAX 8800 performance with a histogram hardware monitor," in *Proc. 15th Annu. Int. Symp. Comput. Architecture*, Honolulu, HI, June 1988.

[10] F. Chow, "A portable machine-independent global optimizer—Design and measurements," Ph.D. dissertation, and Tech. Rep. 83-254, Comput. Syst. Lab., Stanford Univ., Dec. 1983.

[11] J. Cocke and P. Markstein, "Measurement of program improvement algorithms," Res. Rep., IBM, Yorktown Heights, RC 8110, July 1980.

[12] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *Comput. J.*, vol. 19, no. 1, pp. 43–49, Feb. 1976.

[13] B. Currah, "Some causes of variability in CPU time," *Comput. Measurement Evaluation*, SHARE project, vol. 3, pp. 389–392, 1975.

[14] J. J. Dongarra, "Performance of various computers using standard linear equations software in a Fortran environment," *Comput. Architecture News*, vol. 13, no. 1, pp. 3–11, Mar. 1985.

[15] J. J. Dongarra, J. Martin, and J. Worlton, "Computer benchmarking: Paths and pitfalls," *IEEE Comput. Mag.*, vol. 24, no. 7, pp. 38–43, July 1987.

[16] J. J. Dongarra, "Performance of various computers using standard linear equations software in a Fortran environment," *Comput. Architecture News*, vol. 16, no. 1, pp. 47–69, Mar. 1988.

[17] J. S. Emer and D. W. Clark, "A characterization of processor performance in the VAX-11/780," in *Proc. 11th Annu. Symp. Comput. Architecture*, Ann Arbor, MI, June 1984.

[18] J. R. Giles, *Introduction to the Analysis of Metric Spaces*, Australian Mathematical Soc., Lecture Series 3. Cambridge, MA: Cambridge Univ. Press, 1987.

[19] J. H. Griffin and M. L. Simmons, "Los Alamos National Laboratory Computer Benchmarking 1983," Los Alamos Tech. Rep. LA-10151-MS, June 1984.

[20] U. Harms and H. Luttermann, "Experiences in benchmarking the three supercomputers CRAY-1M, CRAY-X/MP, Fujitso VP-200 compared with the CYBER 76," *Parallel Comput.*, vol. 6, pp. 373–382, 1988.

[21] R. N. Ibbett, *The Architecture of High Performance Computers*. New York: Springer-Verlag, 1982.

[22] M. S. Johnson and T. C. Miller, "Effectiveness of a machine-level, global optimizer," in *Proc. SIGPLAN Symp. Compiler Construction*, June 1986, pp. 99–108.

[23] J. L. Kelly, *General Topology*, GTM; 27. New York: Springer-Verlag, 1985.

[24] D. E. Knuth, "An empirical study of Fortran programs," *Software—Practice and Experience*, vol. 1, pp. 105–133, 1971.

[25] D. S. Lindsay, "Do Fortran compilers really optimize?," CMG Transactions, pp. 23–27, Spring 1986.

[26] D. S. Lindsay, "Methodology for determining the effect of optimizing compilers," in *CMG 1986 Conf. Proc.*, Las Vegas, NV, Dec. 9–12, 1986, pp. 379–385.

[27] M. H. MacDougall, "Instruction-level program and processor modeling," *IEEE Computer Mag.*, vol. 7, no. 14, pp. 14–24, July 1982.

[28] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the floating-point performance range," Lawrence Livermore Nat. Lab., UCRL-53745, Dec. 1986.

[29] H. W. Merrill, "Repeatability and variability of CPU timing in large IBM Systems," *CMG Trans.*, vol. 39, Mar. 1983.

[30] Motorola, Inc, *MC68020 32-Bit Microprocessor User's Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[31] Motorola, Inc, *MC68881/MC68882 Floating-Point Coprocessor User's Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[32] B. L. Peuto and L. J. Shustek, "An instruction timing model of CPU performance," in *Proc. Fourth Annual Symp. Comput. Architecture*, vol. 5, no. 7, Mar. 1977, pp. 165–178.

[33] S. Richardson and M. Ganapathi, "Interprocedural optimization: Experimental results," *Software—Practice and Experience*, vol. 19, no. 2, pp. 149–170, Feb. 1989.

[34] R. H. Saavedra-Barrera, "Machine characterization and benchmark performance prediction," Univ. of California, Berkeley, Tech. Rep. UCB/CSD 88/437, June 1988.

[35] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya, "Machine characterization based on an abstract high level language machine," Tech. Rep. UCB/CSD 89/494, Mar. 13, 1989, UC Berkeley, CS Division.

[36] R. H. Saavedra-Barrera and A. J. Smith, "CPU performance evaluation via benchmark prediction," paper in preparation.

[37] M. L. Simmons and H. J. Wasserman, "Los Alamos National Laboratory Computer Benchmarking 1986," Los Alamos Nat. Lab., LA-10898-MS, Jan. 1987.

[38] A. J. Smith, paper in preparation.

[39] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, "Using peephole optimization on intermediate code," *ACM Trans. Programming Languages Syst.*, vol. 4, no. 1, pp. 21–36, Jan. 1982.

[40] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, Oct. 1984.

[41] ——, "Dhrystone benchmark: Rationale for version 2 and measurement rules," *SIGPLAN Notices*, vol. 23, no. 8, Aug. 1988.

[42] J. Worlton, "Understanding supercomputer benchmarks," Datamation, pp. 121-130, Sept. 1, 1984.

Rafael H. Saavedra-Barrera (S'89) was born in Mexico City, Mexico. He received the B.S. degree in electronical engineering from the Universidad Autonoma Metropolitana, Mexico City, and the M.S. degree in computer science from the University of California, Berkeley, in 1988.

He is currently pursuing a doctoral program at the University of California, Berkeley. From 1982 to 1984 he was an Assistant Professor in the Electrical Engineering Department, Universidad Autonoma Metropolitana, Mexico City. During 1985 he was with Logopoiesis, Inc., Mexico City. His research interests include the analysis of computer systems, supercomputers, and programming languages.

Mr. Saavedra-Barrera is a member of ACM and CPSR.

Eugene Miya (S'82–M'83) received the B.S. in mathematics from University of California, Santa Barbara, in 1977.

He is an Aerospace Technologist at NASA Ames Research Center, working on software testing and performance measurement. He is member of IEEE and ACM.

Alan Jay Smith (S'73–M'74–SM'83–F'89), for a photograph and biography, see this issue p. 1630.